

# **Medical Device Software**

## **Verification, Validation, and Compliance**

**David A. Vogel**



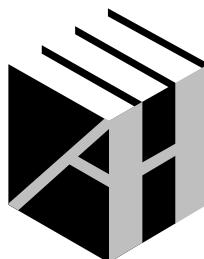
**DVD Included**

# **Medical Device Software Verification, Validation, and Compliance**



# **Medical Device Software Verification, Validation, and Compliance**

David A. Vogel



**ARTECH  
H O U S E**

BOSTON | LONDON  
[artechhouse.com](http://artechhouse.com)

**Library of Congress Cataloging-in-Publication Data**  
A catalog record for this book is available from the U.S. Library of Congress.

**British Library Cataloguing in Publication Data**  
A catalogue record for this book is available from the British Library.

ISBN-13: 978-1-59693-422-1

Cover design by Vicki Kane

© 2011 ARTECH HOUSE  
685 Canton Street  
Norwood MA 02062

All rights reserved. Printed and bound in the United States of America. No part of this book may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without permission in writing from the publisher.

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Artech House cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

10 9 8 7 6 5 4 3 2 1

*To my wife and friend since childhood, Deborah,  
and to my daughters Emily and Audrey*



# Contents

Preface	xvii
The Author's Background and Perspective of Validation	xvii
Acknowledgments	xxi
<b>PART I</b>	
Background	1
<b>CHAPTER 1</b>	
The Evolution of Medical Device Software Validation and the Need for This Book	3
The Evolution of Validation in the Medical Device Industry	3
Building a Language to Discuss Validation	4
Terminology is the Foundation	5
Correct Versus Consistent Terminology	6
Terminology Need Not Be Entertaining	7
Risk Management and Validation of Medical Device Software	8
About This Book	8
Goals of This Book	9
Intended Audience	10
Are You Wasting Time?	12
References	12
<b>CHAPTER 2</b>	
Regulatory Background	13
The FDA: 1906 Through 1990	13
The FDA Today (2009)	16
How the FDA Assures Safety, Efficacy, and Security	17
Quality System Regulations and Design Controls	20
Understanding How Regulation Relates to Getting the Job Done	22
Medical Devices Sold Outside the United States	24
References	25

**CHAPTER 3**

The FDA Software Validation Regulations and Why You Should Validate Software Anyway	27
Why the FDA Believes Software Should Be Validated	27
Therac 25	28
Building Confidence	30
The Validation Regulations	31
Why You Should Validate Software Anyway	34
References	36

**CHAPTER 4**

Organizational Considerations for Software Validation	37
Regulatory Basis of Organizational Responsibility	37
A Model for Quality Systems	39
Roles, Responsibilities and Goals for the Quality System	40
The Structure of the Quality System	41
Quality System Processes	42
Quality System Procedures	43
Thinking Analytically About Responsibility	45
Untangling Responsibilities, Approvals, and Signatures	45
What Happened to the Author?	48
The Meaning of Approval: What That Signature Means	48
So, What Could Go Wrong with a Design Control Quality System?	49
What Happened?	51
Designing Streamlined RR&A Requirements for the Quality System	51
Fixing the Problem: Designing a Value-Added Approval/Signature Process	53
Regulatory Basis for Treating Approvals and Signatures Seriously	54
Reference	55

**CHAPTER 5**

The Software (Development) Life Cycle	57
What Is a Software Life Cycle?	57
Software Validation and SDLCs: The Regulatory Basis	60
Why Are Software Development Life Cycle Models Important?	61
What Do Different Software Development Life Cycle Models Look Like?	62
Waterfall and Modified Waterfall	63
Sashimi Modified Waterfall Model	63
Spiral Model	66
Extreme Programming: Agile Development Models	68
How Do You Know What Life Cycle Model to Choose?	69
How Do Software Development Life Cycles Relate to the Quality System?	70
The ANSI/AAMI/IEC 62304:2006 Standard	73
An Organization for the Remainder of This Book	73
Reference	74

**CHAPTER 6**

Verification and Validation: What They Are, What They Are Not	75
What Validation is NOT	75
Validation and Its Relationship to Verification and Testing	76
Software Validation According to Regulatory Guidance	79
Can Other Definitions of Validation Be Used?	81
User Needs and Intended Uses	82
Software Verification According to Regulatory Guidance	82
How Design Controls, Verification, and Validation Are Related	84
Validation Commensurate with Complexity and Risk	85
Is All Validation Created Equal?	87
Reference	87

**CHAPTER 7**

The Life Cycle Approach to Software Validation	89
Validation and Life Cycles	90
Combined Development and Validation Waterfall Life Cycle Model	91
A Validation Life Cycle Model	93
The Generic or Activity Track Life Cycle Model	95
Life Cycles and Industry Standards	102
Final Thoughts on Selecting an Appropriate Life Cycle Model	103
References	103

**CHAPTER 8**

Supporting Activities that Span the Life Cycle: Risk Management	105
Introduction to Activities Spanning the Life Cycle	105
Risk Management	106
Risk in the Regulations and Guidance Documents	107
ISO 14971: Application of Risk Management to Medical Devices	108
AAMI's TIR32:2004: Medical Device Software Risk Management	110
Risk and the IEC 62304 Standard on Life Cycle Processes	111
IEC/TR 80002-1: Application of 14971 to Medical Device Software	112
The Risk Management Process	112
The Language of Risk Management	113
Risk Management Outputs	114
The Risk Management Plan	114
The Risk Management File	115
Risk Management Concepts and Definitions	115
Risk Management Activities	117
Risk Analysis	117
Qualitative Probability Analysis	122
Ignoring Probability	123
Qualitative Probabilities	123
Risk Evaluation	129
Risk Control	130
Overall Residual Risk Evaluation	134

Summary	140
References	141
<b>CHAPTER 9</b>	
Other Supporting Activities: Planning, Reviews, Configuration Management, and Defect Management	143
Planning	143
Design and Development Planning	143
Why Planning Is Important	144
How Many Plans Are Required?	145
Plan Structure and Content	147
What Does a Plan Look Like?	148
Evolving the Plan	152
Configuration Management	153
Regulatory Background	153
Why Configuration Management?	154
What Goes into a Configuration Management Plan?	155
Defect (and Issue) Management	160
Regulatory Background	161
Why Defect Management Plans and Procedures Are Important	161
Relationship to Configuration (Change) Management	161
Planning for Defect Management	165
Reviews	167
Regulatory Background	167
Why the Focus on Reviews?	168
What Is Meant by a Review?	171
Who Should Be Participating in the Reviews?	172
How Reviews Are Conducted	173
Traceability	177
Why Traceability?	177
Regulatory Background	178
Traceability Beyond the Regulatory Guidance	182
Practical Considerations: How It Is Done	185
Trace Tools	185
Trace Mapping	188
Can Traceability Be Overdone?	189
References	189
<b>PART II</b>	
Validation of Medical Device Software	191
<b>CHAPTER 10</b>	
The Concept Phase Activities	193
The Concept Phase	193
Regulatory Background	194
Why a System Requirements Specification Is Needed	195
Validation Activities During the Concept Phase	196

Make or Buy? Should Off-the-Shelf (OTS) Software Be Part of the Device?	198
The System Requirements Specification	200
Who Is the Intended Audience?	200
What Information Belongs in an SyRS?	201
How Are System Requirements Gathered?	204
Further Reading	205
Select Bibliography	205

## CHAPTER 11

The Software Requirements Phase Activities	207
Introduction	208
Regulatory Background	208
Why Requirements Are So Important	210
The Role of Risk Management During Requirements Development	214
Who Should Write the Software Requirements?	215
The Great Debate: What Exactly Is a Requirement?	217
Anatomy of a Requirement	219
How Good Requirements Are Written	223
Summary	231
References	231

## CHAPTER 12

The Design and Implementation Phase Activities	233
Introduction	233
Regulatory Background	234
Validation Tasks Related to Design Activities	236
The Software Design Specification (Alias the Software Design Description)	236
Evaluations and Design Reviews	239
Communication Links	239
Traceability Analysis	240
Risk Management	246
Validation Tasks Related to Implementation Activities	247
Coding Standards and Guidelines	248
Reuse of Preexisting Software Components	248
Documentation of Compiler Outputs	249
Static Analysis	250
References	251

## CHAPTER 13

The Testing Phase Activities	253
Introduction	253
Regulatory Background	253
Why We Test Software	255
Defining Software Testing	256
Testing Versus Exercising	257
The Psychology of Testing	258

Levels of Testing	260
Unit-Level Testing	261
Unit-Level Testing and Path Coverage	263
McCabe Cyclomatic Complexity Metric and Path Coverage	263
Other Software Complexity Metrics and Unit Test Prioritization	267
Integration-Level Testing	267
Device Communications Testing	269
System-Level Software Testing	272
System-Level Verification Testing Versus Validation Testing	274
Testing Methods	275
Equivalence Class Testing	276
Boundary Value Testing	279
Calculations and Accuracy Testing	282
Error Guess Testing	286
Ad Hoc Testing	287
Captured Defect Testing	288
Other Test Methods	289
Test Designs, Test Cases, and Test Procedures	290
Managing Testing	295
The Importance of Randomness	295
Independence	296
Informal Testing	297
Formal Testing	298
Regression Testing	300
Automated Testing	302
Summary	303
References	304
Select Bibliography	304

**CHAPTER 14**

The Maintenance Phase Validation Activities	305
Introduction	305
A Model for Maintenance Activities	308
Software Release Activities: Version $n$	309
Collection of Post-Market Data	312
Process and Planning	313
Sources of Post-Market Data	313
Analysis	315
The Maintenance Software Development Life Cycle(s)	318
Software Development and Validation Activities	320
Software Release Activities: Version $n + 1$	321
References	321

**PART III**

Validation of Nondevice Software	323
----------------------------------	-----

**CHAPTER 15**

Validating Automated Process Software: Background	325
Introduction	325
Regulatory Background	326
Nondevice Software Covered by These Regulations	330
Factors that Determine the Nondevice Software Validation Activities	332
Level of Control	332
Type of Software	334
Source of the Software	334
Other Factors That Influence Validation	335
Risk	336
Size and Complexity	336
Intended Use	336
Confidence in the Source of the Software	337
Intended Users	337
Industry Guidance	340
AAMI TIR36:2007: Validation of Software for Regulated Processes	341
GAMP 5: Good Automated Manufacturing Practice	341
Who Should Be Validating Nondevice Software?	342
Reference	343

**CHAPTER 16**

Planning Validation for Nondevice Software	345
Introduction	345
Choosing Validation Activities	346
Do-It-Yourself Validation or Validation for Nonsoftware Engineers	347
The Nondevice Software Validation Spectrum	349
Life Cycle Planning of Validation	350
The Nondevice Software Validation Toolbox	352
Product Selection	354
Supplier Selection	354
Known Issue Analysis	355
Safety in Numbers	355
Third-Party Validation	356
Output Verification	357
Backup, Recovery, and Contingency Planning	358
Security Measures	359
Training	360
The Validation Plan	360
Reference	361

**CHAPTER 17**

Intended Use and the Requirements for Fulfilling Intended Use	363
Introduction	363
Intended Use	364
Why It Is Necessary to State Intended Use	364
Intended Use and Validation of Nondevice Software	365

Contents of a Statement of Intended Use	365
Determining Intended Use	366
Requirements for Fulfilling the Intended Use	369
Requirements for Custom-Developed Software	369
Requirements for Acquired Software	370
Information Content of Requirements	370
Example: Intended Use and Requirements for Validation of a Text Editor	372
<b>CHAPTER 18</b>	
Risk Management and Configuration Management of Nondevice Software Activities that Span the Life Cycle	375
Risk Management	375
Applying the 14971 Risk Management Process to Nondevice Software	375
Harm	376
Risk, Severity, and Probability	378
Managing the Risk	382
Controlling the Process to Reduce Risk	383
Risk Acceptability	383
Detectability	387
Configuration Management for Nondevice Software	387
Why Configuration Management Is Important	388
Configuration Management Planning	389
Configuration Management Activities	391
References	392
<b>CHAPTER 19</b>	
Nondevice Testing Activities to Support Validation	393
Why Test—Why Not To Test	393
Testing as a Risk Control Measure	395
Regulatory Realities	395
Testing Software That Is Acquired for Use	396
IQ, OQ, and PQ Testing	397
Validation of Part 11 Regulated Software	399
Summary	400
<b>CHAPTER 20</b>	
Nondevice Software Maintenance and Retirement Activities	401
Maintenance Activities	401
Release Activities	402
Post-Release Monitoring	403
Risk Analysis and Risk Management	404
Security	405
Retirement of Software	406
About the Author	409
Index	411

# Preface

We are all products of our past. Our opinions cannot help but be influenced by other's opinions. Our experiences personalize our opinions based on what we have seen that is good and bad, what works and what does not. Software validation is subject to opinion. It is part science, part engineering, part psychology, and part organizational management. Exactly what validation is and how it should be accomplished is subject to opinion because it is not an exact science.

In this preface, I share some perspective on my insights into software validation and how they came to be what they are. It is a rather loose collection of thoughts that I feel are important to convey, but do not fit cleanly into the more technical sections of the book.

## The Author's Background and Perspective of Validation

I don't trust software.

Maybe it's just me. Maybe I've become a product of my profession, or maybe my professional development has been driven by my suspicion that software—any software—is probably going to fail in some way when I use it, and probably when I need it most. I'm rarely disappointed in that regard.

I seem to notice a problem with every new software package I install within the first few minutes of using it. Even if I have used the software for years, I seem to run into problems whenever I try to use an advanced feature for the first time. As shown in Figure P.1, I have even run into problems with our office adding machine, which I am sure has to be controlled by software. This figure shows that the sum of the short list of numbers differs between the display and the printed tape.

I am a software engineer. There, it's out. I've written a lot of software in my career. I've written software with bugs in it. I've studied computer science and software engineering and their application to biomedical instrumentation at respected universities for years. I still don't trust software.

As a graduate student, I wrote software that controlled an apparatus that collected data for my research. I slapped it together hurriedly, and paid the price when I found it had problems. That cost me time, but few others knew about the problems, were affected by them, or even cared. Later as part of a research assistantship, I wrote software to automate the experiments of my advisor and other postdoctoral researchers. Now others were affected and did care (a lot). It still cost me time and dented my reputation. It got me to think about how I could improve my software development process to make the software more reliable, understandable, and maintainable, so that I could get out of graduate school while still a young man.

I did get out, eventually, and took a job with a well-known company that manufactured patient monitoring systems for hospital coronary care units. It was already



**Figure P.1** Software defects may plague devices not normally associated with software.

too late for me. I had already developed some skepticism about the reliability of software, and now was in a job to develop software upon which peoples' lives depended. It is not being overly dramatic to say that I lost a lot of sleep (and developed some nervous habits) over that software. As a group, we developed a very systematic methodology for testing that software. Thankfully, executive management saw the value of that testing, and allowed us to pour man-years of effort into creating and executing the test procedures. It was a far cry from what we call “validation” today, but it found problems. Additionally, it

- Provided a basis for metrics on how bug-free the software was;
- Established a base for testing later releases of the software;
- Made us aware how imperfect our development process was;
- Built confidence in the software that was released once it passed the test suite.

This took place in the days before the FDA had “discovered” software in medical devices and there were no regulations or guidance documents to point us in the right direction. As a result of our testing experience we began to focus on a better development process to keep the defects from getting into the software to start with.

I moved on to another (consulting) job and my software skepticism was further fueled by witnessing the same kinds of unreliability of software in the defense industry, automotive industry, process control industry, oil and gas industry, and lab instrumentation industry. This was in the early days of microprocessors, and we had a saying around the office that “World War III will be started by a bug in some microprocessor code.” I’m sure if I took the time to research this, I could find documented tales about how close we have come to this being a fact.

Eventually, I founded my own engineering services company (Intertech Engineering Associates, Inc.) to develop and validate software specifically for the medical device industry. As a company, we continued to emphasize the importance of testing to our clients. We also became strong proponents of requirements specifications and design documentation. Part of this affinity to requirements had to do with the self-preservation of our business. We learned that if we were to develop products for manufacturers under contract, we needed detailed requirements to be sure we met client expectations, and so all parties knew when the project was done. The value of requirements for testing inputs had long been evident, but, in reality, it was the business need that made it clear to us that there was great value in hashing out requirements before we spent time writing software.

The value of good requirements and design documentation for the maintenance of the products we designed became clear after we had completed the first or second project. We were flooded with calls from the clients because our design documents were so sparse. We realized that we needed to improve on documentation to survive. We were having trouble finding billable time on the next project because of the time we spent verbally maintaining the last project.

Eventually, the IEEE released their software standards (or maybe we just became aware of them). The FDA became software-aware and provided the industry with the General Principles of Software Validation (which will be referred to as the GPSV in this text). Both are excellent and valuable. Both have their problems. As a company, we got behind the GPSV, set up our own quality system that focused on software development and validation, and helped our clients validate their software along the GPSV guidelines.

The rest, as they say, is history. Intertech is in its 29th year of providing engineering and consulting services to the medical device industry. We have continued to improve our own quality processes. We struggle with it just as our clients do. Our experience continues to shape and refine our opinions about which development and validation techniques work, and which do not. The reason for investing the time in writing this text is to share that experience with you the reader so that it won't take you 29 years to come to similar conclusions.

The theme of this text is to remind the reader repeatedly why validation is important and why each activity is important and should add to the confidence in the software being validated.

It is my sincere hope that this book will play some role in convincing readers to take medical device software validation seriously and make it a process that makes medical devices safer. Our own lives and the lives of our families, friends, and coworkers will at some point depend on the devices that you, the reader, will develop and validate. None of us wants to worry about device safety when we see a device connected to a loved one.



# Acknowledgments

I owe a special thanks to Meredith Lambert, who prepared all the figures for this book, and whose creative skills saved us all from a book full of stick figures.

I also am indebted to my reviewers, especially Jeremy Jensen and John Murray for their careful reading and thoughtful comments on the content of this book. Thanks are also due to my editor Lindsey Gendall who tread carefully in reminding me of deadlines—often.

The content of this book is the result of years of experience at Intertech Engineering Associates, Inc. That experience came through the patient discussions, debates, and disagreements over the details of how to get the jobs done. Thank you to all employees and clients—past and present—for your patience. Jennifer Bernazzani of Intertech deserves a special recognition for her support in reviewing the proofs of this book.

My family also deserves acknowledgment for enduring my “absence” from family events as the writing schedule took priority. Thank you, Debbie, for your patience and encouragement along the way.



PART I

# Background



# The Evolution of Medical Device Software Validation and the Need for This Book

Let us begin this text by taking a look at the relatively short history of software in the medical device industry to gain some understanding of just why the regulatory definition of validation includes the activities that it does. The evolution of the medical device industry, the software disciplines, and the regulatory understanding of software in medical devices all played a role in how validation is currently perceived.

## **The Evolution of Validation in the Medical Device Industry**

Since founding our company in 1982, we have watched software development in the medical device industry evolve. In 1982, we were writing software in assembly code. An ambitious product contained thousands of lines of code. Development teams typically were made up of one to three engineers. Today, ambitious products can contain millions of lines of code, embedded in multiple microprocessors, with development and/or validation teams of dozens of engineers spread geographically around the world. Clearly, the industry's processes for software development and validation needed to evolve with the increasing dependence on software that controls medical devices.

Our company works on an average of about five devices a year for a variety of clients. In addition to that, we consult with a dozen or two clients a year in an advisory capacity on topics related to product development or validation. This gives us a broad perspective on how companies in the industry think, act, and portray themselves to the regulatory agencies. We know from firsthand experience that many companies and individuals in the industry feel that design controls are a waste of time and impede product development. Many feel that their validation budgets are disproportionately large or that their validation efforts are not effective enough at finding defects to legitimize the cost. Software developers seem to be in a constant tug-of-war with the compliance and quality professionals. Too many developers spend too much time coming up with clever ways to get around the quality system, or to prove to themselves that the efforts of the quality team are worthless.

Likewise, too many quality professionals spend too much time getting too involved in the development process requiring unnecessary formal reviews and documentation to create the appearance of a controlled process. Imagine if only half this amount of creativity could be applied to working together to improve the process and actually improve the quality of the software.

Can validation activities actually shorten the development life cycle? Until someone figures out a definitive experiment for determining the answer to this ques-

tion, we will have to settle for opinion and anecdotal evidence. Based on my personal experience and anecdotal evidence, my opinion is that a methodical, systematic development process interwoven with an equally systematic validation process gets products to market faster. We all know there are developers and managers who think they can beat the system by first writing the software then retro-documenting to make it look like a controlled design process was followed. We know there are companies who think they can hire compliance experts to “sell” them a quality system in a binder, then ignore the binder until they near completion of the product. We know there are companies that measure the quality of their validation documentation by how high it stacks on the table when completed. We know there are developers and managers who have come from outside the medical device industry and are sure that none of these regulations or guidelines applies to them.

We know there are differences of opinion in the industry. There are even differences of opinion within our own company. Sometimes those differences lead to better ways to do things. Sometimes they just get in the way. I do know from many years and many clients, that there is a false perception of efficiency in not following a process.

Many of the activities that keep a systematic process on course are validation activities. In later chapters as we examine the full range of activities that are considered to be part of validation, it will become clear that validation and development activities are so closely interwoven it's sometimes difficult to distinguish between the two.

## **Building a Language to Discuss Validation**

A common problem in software validation is miscommunication because of loosely defined terminology. It is not unusual to participate in a meeting in which a number of the participants are using the same terms in the discussion, but each has a totally different understanding of what those terms mean. This can lead to extended debate, misunderstanding, and poor decision-making. As an example, imagine a meeting that is called to assign resources for integration testing for a device. “Integration testing” could, and most likely would, mean different things to different participants in the meeting based on their past experiences with other device validation programs. It is not hard to imagine the meeting turning into a debate over whether certain individuals are qualified to perform the integration testing, and how long that testing should take. Most of the debate and wasted time could be attributed simply to the fact that the participants had totally different concepts of integration testing in mind. Given a common understanding of what integration testing means, there is a good likelihood that the participants would have been mostly in agreement.

Going a step further, it is not at all unusual for validation professionals and developers to use terminology without a clear understanding of what the terms really mean. This problem seems to be human nature, and is not at all limited to terminology related to software validation for medical devices. The problem also is not limited to rookies and the uninformed. Experienced professionals, consultants, and “industry experts” are guilty of this too (including myself). People new to the profession will interrupt to ask what the terminology means. Experienced professionals

are so accustomed to using the terminology they may not have stopped to think about what it actually means since they themselves were new to the profession.

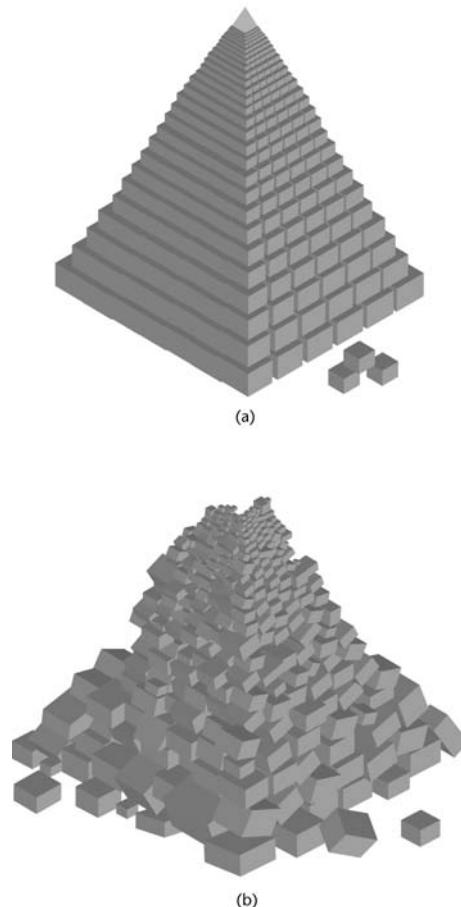
At the next meeting in which the discussion is being prolonged by a mismatched understanding of the terminologies, ask the meeting attendees, “What exactly do you mean by XXX?” (where XXX is a term of your choice that is relevant to meeting). Don’t be surprised if the response is something like, “Well, we *all kind of know* what XXX means.” Don’t be intimidated. That just means, “*None* of us knows *exactly* what XXX means.” In my experience, one usually finds that the participants around the table have quite different definitions for the terms, and those differences often are at the heart of the debate at hand. A new deliberation over the definition of the term XXX may replace the original debate.

### Terminology is the Foundation

Why does this happen? There are some reasons that are easy to fix. A person may simply be inexperienced, or may have had experience at a different company or even in a different industry in which the terminology is used differently. A person could simply be wrong. These are simple retraining issues. Regardless, it is simply human nature for us to use our day-to-day terminology as building blocks of our language *assuming* we all have a common understanding of their meaning. The first panel of Figure 1.1 represents how we use those building blocks to put together increasingly complex sentences that are themselves the building blocks of ideas that are embedded in policies, processes, procedures, plans, and so forth. Unfortunately, as the second panel of Figure 1.1 depicts, when those basic building blocks are weakened because they are not well defined (misunderstood, inconsistent, overlapping, or with gaps) or if they are not adequately communicated, the entire hierarchy of thought based on those terms is weakened.

As important as they are, well-established definitions are not a substitute for good processes and procedures in an organization. Rock-solid technical definitions in the design documents for a device do not guarantee a good design. Back to our analogy, defective buildings can be built of perfect building blocks. However, clearly communicated definitions for the terminology we use will make it easier and more likely to find defects in our *higher-order thinking* simply because we do not base so much of our review of documents and our evaluation of results on *assumption*.

There is usually some level of reluctance to start a deliberation over the definitions of terminologies in a product development or validation program. That is understandable; after all, most of us show up at work every day to move a day closer to getting a device to market. Taking time to have a debate over terminology is not perceived as advancing the schedule by much. Even worse, redefining terminology could result in rework, setting the schedule back. However, ignoring a common understanding of our language could lead to a false impression of making progress. To further the building block analogy, would we continue to build a multistory building on top of a weak foundation simply to stay on schedule? Eventually buildings and languages run into problems without well-engineered foundations—and that ultimately takes time out of the schedule and money out of the budget.



**Figure 1.1** (a, b) Well-ordered terminology allows us to build strong understanding.

What *can* we do about this? A good starting point is not to ignore the problem, and not to intimidate others from questioning terminology. Taking charge of terminology early on, whether in writing regulations, industry standards, corporate standard operating procedures, project plans, requirements or design documents avoids all that messy rework later on when it is discovered that the team is not all speaking the same language.

### Correct Versus Consistent Terminology

Some have expressed the opinion that spending a lot of time debating the *correct* definitions of terms is a waste of time. With some exceptions to be discussed shortly, we can avoid philosophical debates about the rightness and wrongness of terminology. All that is needed is to keep a running glossary or table of definitions for the intended meaning of the terminology in the context in which it is currently used. As long as the usage of the terminology is consistent, clear communication is established. It *would not* be wise for every document within an organization to have its own definition for each term; some common sense is required in managing

terminology. Similarly, once clearly defined terminology is established, it would not be wise to modify the terminology without some controls. Too many documents may end up depending on that terminology, and changing the definitions may change the meaning in those documents in which the terminology is used.

There are exceptions where right and wrong should be debated before committing terminology definitions to writing. The exceptions that come to mind are:

- When claiming compliance with regulations, regulatory guidance, or industry standards (let's call them *governing documents* here even though compliance with some in the list is voluntary), any terminology used in common with those governing documents should use the same definitions for terminology. This makes it easier to determine whether the intent of the governing documents has been fulfilled, and will make it easier to explain to a potential auditor or inspector how you have fulfilled the requirements of the governing document.
- When a governing document is written by a governing body that will have industry-wide exposure, careful consideration must be given to the *exact* definitions of terminology, and those definitions need to be *correct*. In this context, the governing body could be a legislative body, standards committee, or industry workgroup. “Correct” here means that the terminology correctly conveys the intent and the entire intent of the governing body without ambiguity, inconsistency, or overlap with other terminologies used by the governing body. The reason for the emphasis on correctness is that such definitions (or lack thereof) will be read, interpreted, and debated by the entire industry. The less that is open for interpretation, the less confusion there is likely to be, and the more likely that the industry will fall in line with the intent of the terminology.
- Within the hierarchy of documentation of a company's quality system, terminology that is used higher on the hierarchy tree deserves more debate for correctness than that in lower level documents. This is for two reasons. First, when the terminology is related to a regulatory requirement or industry standard with which the company has chosen to comply, getting terminology right centralizes the debate over interpretation of the governing requirement. Further, it promotes understanding whether the company terminology will comply with the governing intent. Second, for the same reason that governing documents need careful scrutiny for correctness, quality system terminology should be debated for correctness simply because it will impact all lower-level documents that rely on that terminology.

## Terminology Need Not Be Entertaining

Clarifying definitions is a requirement for clear communication of validation processes, product requirements, and validation artifacts. Nonetheless, it is common for those on the receiving side of a critical review of terminology to be defensive and reluctant to continue the review. It is also common to hear complaints that using consistent terminology makes documents boring to read and stifles the creative instincts of the writer. Exactly. Nobody ever said that writing done to record our

validation processes and procedures, or to document requirements, specifications and designs was supposed to be entertaining, did they? Creating them should *not* be an exercise in creative writing. Save the creativity for product design and test methodology. Creative writing for technical documents always leads to confusion when the writer uses multiple terms to describe the same thing simply because it was “boring” to use the same terms over and over or may be perceived as not entertaining enough for the reader.

With that as a background, perhaps it will be understandable why this book spends some time on terminology. The terminology used here may not agree with the way you or your company use the same terms. However, by carefully defining terms as they are used, at least we will be able to clearly communicate the rationale behind how the terms are used.

## Risk Management and Validation of Medical Device Software

There is lengthy coverage in later chapters of this book devoted to risk management, so it is not necessary to say too much here. I spent several years with a workgroup of industry experts that authored AAMI’s *Technical Information Report AAMI TIR32:2004 on Medical Device Software Risk Management*. After that amount of time spent deliberating on one subject, my viewpoints and opinions couldn’t help but be changed. FDA guidance documents and industry standards describe risk management for medical device software as a supporting process to the software development and validation processes. This book will be consistent with that viewpoint so as not to inject further confusion into the industry.

It is interesting to consider, however, that all of the activities associated with software validation are done to manage the risk of harm related to software failures. One could, with equal justification, say that validation is a supporting process of risk management rather than the more commonly accepted inverse.

Why is this worthy of note? Understanding this very close relationship between the two concepts helps us understand why we do what we do in software validation. We do it all to reduce the risk of harm associated with medical device software.

At Intertech, at some point in the new employee training process, we let the newcomers know how important their work is. The devices we work on may well end up being used on us or on our loved ones. We are not in business to shuffle validation paperwork. We are in business to be sure that the devices, and specifically the software that runs those devices, is safe. This is a sobering thought, and helps us stay focused on validation activities that reduce risk, not activities that generate paperwork for the sake of generating paperwork.

## About This Book

This book is about the validation of software for the medical device industry. The industry struggles with the exact definition of verification and validation. However most would agree that the validation activities vary depending on the type of software and the way the software is to be used. This book is broken into three major

parts. Part I provides some historical and technical background of software validation that applies to all software validation. Part II deals specifically with the validation of software that is an embedded part of a medical device, or software that is itself a medical device. Part III deals with software that is *used* in the design, production, maintenance, and control of quality of medical devices; that is, nondevice software whose validation is also regulated. The approaches and details of validating the two very different types of software share some similarities, but are quite different, especially for the nondevice software that is acquired for use.

There are numerous textbooks that deal with software testing. There also are numerous books that deal with regulatory compliance of medical devices. There are few books that deal with software validation in the context of the regulated medical device industry. That is the void that this book will address.

As of this writing, the author has logged almost three decades of experience designing, developing, and validating medical devices and life science instrumentation. During that time, the term “validation” was introduced to the medical device industry in language found in the United States Code of Federal Regulations (CFR) in 1996. The industry’s concept of what validation means has evolved continuously since the term was introduced. The Food and Drug Administration (FDA) has provided some guidance in understanding their expectations for validation in the *General Principles of Software Validation*, [1] which was initially released in 1997 and rereleased in 2002. More recent evolution of the industry’s understanding of software validation has been driven by industry standards groups, trade literature, workshops, seminars, webinars, past experience, and word-of-mouth (folklore). In fact, recent opinions from the FDA in the United States [2] make it clear that medical device manufacturers are expected to rely on industry sources for current information and opinion on validation. Although this seems arbitrary and inefficient on the surface, it does leave the industry with sufficient freedom to innovate and improve validation methodologies within the constraints of the regulation. As one would expect from uncoordinated evolutionary advances, the industry’s understanding of the topic is sometimes more driven by opinion and belief of what is required by the FDA and other regulatory bodies, than it is by factual information about regulatory requirements, what really works, and what adds value to the validation process.

## Goals of This Book

Although many have tried, and continue to try, software validation defies a single template, checklist, or one-size-fits-all approach. One goal of this book is to share with the reader the author’s own opinions and beliefs about software validation. Another, and more important goal, is to educate the reader about *why* the various validation activities discussed in the text build confidence in the software being validated. In other words, understanding *why* we do what we do in software validation is a more important lesson to be delivered by this book than the details of *how* we do what we do.

The intent is to arm the reader with an understanding of software validation activity concepts, and an understanding of why, when, and how each of those activ-

ties contributes to a validated state for software in the medical device industry. Once so armed, the reader is better equipped to think critically about undertaking the next software validation task rather than relying on a set of forms or checklists that may not be appropriate for that particular validation project.

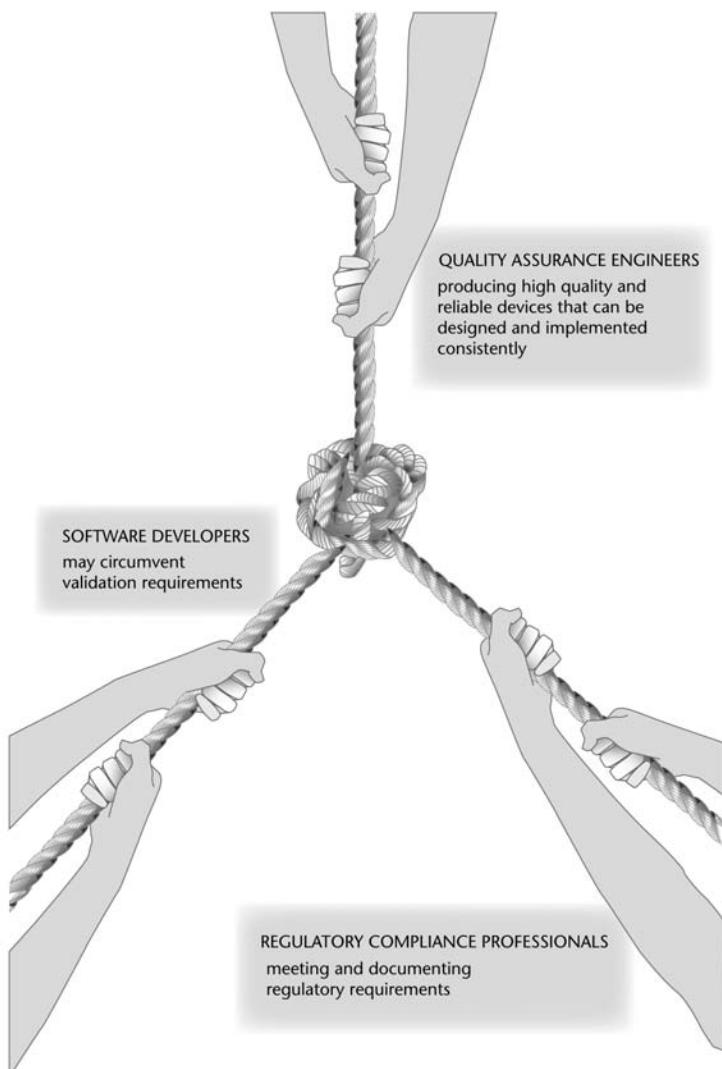
Validating software in the regulated medical device industry cannot ignore the realities of regulations, regulatory guidance documents, industry standards, industry technical information reports, and corporate policies, processes, and procedures (i.e., standard operating procedures or SOPs). A third goal of this book is to contribute to the understanding of how corporate SOPs can make use of regulation, regulatory guidance, standards, best practices, and other bodies of knowledge.

## Intended Audience

Part II of this book, which deals with the validation of software that is an embedded part of a medical device, is written for an audience that includes quality assurance engineers, regulatory compliance professionals, and, yes, even software developers. Each of these groups of professionals has a different view of validation and a different set of objectives that they want validation to achieve for them. Oftentimes, as Figure 1.2 depicts, the process is more like a game of tug-of-war than a team effort with everyone pulling in the same direction.

Quality professionals simply want high quality and reliable devices that can be designed and implemented consistently over time and across the company's product line. They generally have little responsibility for development cost or schedule (nor should they). Compliance professionals are primarily concerned with meeting regulatory requirements and whether the activities that meet those requirements have been documented adequately for them to present it to auditors and regulatory inspectors in the future. All too often, their focus is on the regulatory documentation, and not so much on whether the processes that underlie the documentation are meeting the regulatory intent to assure safe and effective devices. Unfortunately, the software developer's view of software validation is not always a complimentary one. Some developers make it a personal goal to come up with creative ways to circumvent any validation requirements that are their responsibility. A major goal of this book is to share the vision of validation that each group has with the other two groups. Perhaps an understanding of why each validation activity is important will lead to more cooperation and tolerance among the groups. Cooperation among the three major stakeholder groups is a key ingredient for success in any validation program.

Part III of the book deals with the validation of software used to control the development, manufacture, and quality of medical devices (i.e., nondevice software). The skills and tools for validation as described in Part III are similar to the skills and tools described in Part II with one major difference. Much of the software in this category (perhaps even most of the software in this category) is software that is purchased "off the shelf." Consequently, many of the validation activities that take place during the *development* of software are not reasonably applicable to software that is simply *acquired for use*. The role of the software developer is diminished or completely absent, and a new potential validator emerges. The user himself or



**Figure 1.2** Medical device development as a tug-of-war among professionals.

herself may end up bearing the majority of the responsibility for validation of software that is acquired for use. Consequently, *users* of nondevice software are yet another intended audience of this book, especially Part III. Much of the discussion in Part III is written from the perspective of these “user/validators.” An understanding of why each of the validation activities is recommended is just as critical for each of the audience groups for this type of software as is for the embedded medical device software.

Finally, corporate management should take some time with this book to understand *why* the validation activities are important. Too often a basic lack of understanding of validation results in management’s disbelief in the resources required to validate software. Perhaps an understanding of what the activities are, and why they are important will get management also pulling in the same direction.

## Are You Wasting Time?

I have said it before about software validation, and will probably say it again thousands of times: “If you feel like you are wasting your time, you probably are.” That is not to say that validation is a waste of time. If you don’t have the sense that your validation activities are finding defects, or building your confidence in the software and the way it was designed, developed, and implemented, then *there is something wrong with your validation process*. If validation activities are not finding defects and building confidence, then *it is* a waste of time. That does not mean you should forget about validation; it means you should fix your validation process.

Most commonly, I hear clients complain that validation is a waste of time, or is slowing them down simply because *they are not following their own process*. Software developers who are spending time retro-documenting requirements and designs *after* the software is written probably *are* wasting their time, and *it is* holding up completion of the software. The time for documentation and review of requirements and designs is *before* implementation. That is when there is *value* in the process, not when these documents are written after the software to fulfill an internal “quality” audit or regulatory requirement. Following the process may actually have reduced the implementation time by cutting down on the “random walk” implementation process of trial and error.

Nine times out of ten when I hear these complaints, the client’s development and validation processes correctly identify validation activities that are to be performed, and in the right order. The process simply wasn’t followed or looked at—until someone started to write up the regulatory submission and started asking where the validation artifacts were.

There are no validation activities that will be suggested or discussed in this book that are a waste of time. Each activity is there for a reason. That reason will have something to do with improving the quality or confidence in the software. My goal in writing this book is to make it clear to the reader where the value is in each activity.

## References

- [1] *General Principles of Software Validation; Final Guidance for Industry and FDA Staff*. Center for Devices and Radiological Health, United States Food and Drug Administration, January 11, 2002.
- [2] United States of America vs. Utah Medical Products, Inc. United States District Court, District of Utah, Central Division. Case No. 2:04CV00733 BSJ. Defendants’ Pretrial Brief, September 19, 2005. Appendices A and B.

# Regulatory Background<sup>1</sup>

Software engineering for the medical device industry (MDI) is not the same as software engineering in other industries such as the consumer electronics industry. Product life cycles in the consumer markets are often measured in months. Time to market often is more important than the reliability of the software. In the medical device industry, it is not unusual for products to have product lifetimes exceeding 10 years. The safety and efficacy of medical devices trump time to market. That prioritization of safety and efficacy over speed and profitability is the responsibility of the FDA.

The medical device industry is a regulated industry. There is a lot of confusion among lay people, and, unfortunately even within the medical device industry about what that regulatory power is and what it means to us as medical device developers. For now, it will suffice to say that the regulatory oversight reaches much farther than a review of test results in a manufacturer's premarket submission.

Regulatory oversight also governs *how* the device was developed, not just *what* it turned out to be. The reason, as we will see, is that there is a belief that a well-planned, systematic engineering process produces more reliable devices, especially if software is a component of the device.

This is not a book on regulatory compliance, the history of the FDA, or on the history of the international counterparts of the FDA. However, it is an interesting history, and it is important to know the history to understand the reasons (i.e., the whys) behind how the regulatory system operates. *That* is important to help understand reasons behind some of the technical activities discussed in the remainder of the book.

## The FDA: 1906 Through 1990

The FDA, like most other federal agencies, has been reactive in the way it was formed and the way its regulations have evolved and grown over time. Consider for a moment why this has to be the case.

- It is impossible for any governmental agency to be so omniscient and predictive as to write regulation in advance of any and all situations that may evolve over time. Consider the technology changes we have seen in our lifetimes. What legislator or regulator could foresee the advances in science and technology with microelectronics, software, and genetic engineering to *proactively* write meaningful legislation to control it in 1906?
1. Many thanks to Dr. Lita Waggoner-Pogue and Ms. Amelia Gilbreath (Cenotti) who coauthored the following publication with me on the historical background of the FDA. Much of the early part of this chapter is based on that research, "Which Rules Must We Follow?", Designfax, May 2004.

- It is similarly impossible for legislators and regulators to predict “what could go wrong” in the future so it could be alleviated by regulation. Some of this is a corollary to our first bullet. It is difficult to know what can go wrong with a technology until we have an opportunity to apply it to medical devices and see what can go wrong in the intended use environment. When things go wrong with medical devices it can be because of the truly unforeseen and unpredictable. Unfortunately, it can also be due to those in the industry who cut corners, or take advantage of loopholes. Regardless of the cause, new or expanded regulation addresses the issue. This is how the FDA evolves to meet the ever-changing needs of the medical device industry. This is also what causes some in the industry to complain that the FDA is a moving target when it comes to regulatory compliance.

The FDA had its roots in the Food and Drugs Act of 1906. This legislation was a reaction to increasing public concern at the time over poisonous food additives and ineffective patent medicines. However, it was Upton Sinclair’s book *The Jungle* and its graphic description of the unsanitary meat packing practices of the time that was the tipping point for Congress to act. To regulate this legislation, the Food Drug and Insecticide Administration (whose name was shortened in 1930 to today’s Food and Drug Administration or FDA) was created as an independent regulatory agency. The Food and Drugs Act of 1906 did not address medical devices, although they certainly existed at the time. Consequently, the Food Drug and Insecticide Administration had no power to regulate medical devices, and the power it did have over food and drugs was limited to after-the-fact seizure of misbranded or adulterated products.

In 1938, Congress passed the Food Drug and Cosmetic Act (FDCA). Again, it was a time of public dissatisfaction with ineffective and sometimes unsafe devices, drugs and cosmetics. Legislators recognized the inadequacies of the Food and Drugs Act of 1906, and debated for almost 5 years a rewrite of the Act. As before, Congress was finally prompted to act in 1938 because of a large-scale drug related disaster. The year before, the Elixir of Sulfanilamide killed over 100 people. The elixir contained as an ingredient diethylene glycol (as in antifreeze) which was known even then to be poisonous.

The Food and Drugs Act of 1906 dealt primarily with misbranded (i.e., unproven claims) and adulterated (potentially unsafe) foods and drugs. The FDCA of 1938 expanded regulatory control to include cosmetics and therapeutic devices. It also established a requirement for new drugs to be proven safe *prior* to release to the marketplace. The FDCA was the first legislation that authorized factory inspections and that allowed for court injunctions in addition to the after-the-fact seizures and prosecutions [1]. The focus even with the FDCA was on safety. Methods for determining efficacy were slow in developing, and legislation was consequently slow to develop for efficacy.

In 1962 yet another tragedy (this time in Europe with the drug thalidomide), prompted Congress to pass legislation requiring drugs to get approval from the FDA prior to release to the market. This established the foundation for today’s premarket approvals (PMAs) that now apply to devices as well. Still, the FDA control over medical devices was weak, and in some cases, the Agency sought court orders to

have certain devices declared equivalent to a drug so that it could apply their regulations requiring effectiveness to devices [2].

The medical device industry flourished with minimal regulation even as the number and complexity of medical devices increased rapidly. In 1976, medical device tragedy struck once again as thousands of women were injured by the Dalkon shield intrauterine device. Legislative response was the creation of the Medical Device Amendments (MDA), which finally required premarket approval for medical devices. Furthermore, the agency was given the power to withhold approval for devices they determined were ineffective or unsafe, or to recall devices that were approved but subsequently found to be unsafe or ineffective [3]. Devices deemed to be *substantially equivalent* to those on the market prior to the enactment of the MDAs in 1976, whose safety and effectiveness had already been established, were granted special streamlined premarket notification submission requirements detailed in Section 510(k) of the Medical Device Amendments of the Federal Food Drug and Cosmetic Act. Additionally, these amendments expanded the FDA's control of devices to include diagnostic as well as therapeutic devices. The amendments also included requirements for medical device manufacturers to follow quality control procedures as detailed in the first good manufacturing practices (GMPs).

That brings us to the mid-1970s, the golden era of the minicomputer and the early days of the microprocessor. Medical devices experienced a quantum leap in complexity thanks to the power of software for controlling devices, managing data, and making decisions. Additionally, many of the mundane monitoring and control functions of electromedical devices began migrating from dedicated hardware to software. But software, as it turns out, is one of those technologies not anticipated by prior regulation, and was waiting for its disaster to prompt regulatory action. The disaster came in the late 1980s when a number of cancer patients received massive X-ray overdoses (some experiencing entrance and exit burn symptoms) during radiation therapy with the Therac-25 linear accelerator. This lead to a number of investigations, perhaps the most thorough of which was that of Leveson and Turner [4], which was rich with identified ways software could go wrong. Inadequate testing, dangerous code reuse, configuration management issues, inadequate manufacturer response, and failure to get to the root cause of the problem were among the leaders of the problems identified by Leveson and Turner. However, pretty much anything that could go wrong with software did go wrong on the Therac-25—and it was considered reasonably and professionally engineered by the standards of the day.

Needless to say, Therac-25 was an eye-opener for the FDA and legislators and resulted in a flurry of legislative and regulatory activity. The Safe Medical Device Act of 1990 required closer medical device tracking and postmarket surveillance and tracking of implantable medical devices. Hospitals and nursing homes that used medical devices were now required to report device failures that did or could have resulted in serious injury or death.

Subsequent to the enactment of the Safe Medical Devices Act, a series of draft guidance documents were released from the FDA to explain their views on how the Act applied to software in medical devices. These documents all have been released by the FDA since the Therac 25 incident. These guidance documents are available on the FDA's website ([www.fda.gov](http://www.fda.gov)). They are also included on the

accompanying DVD, and current versions are always mirrored on our website (<http://www.inea.com/FDA.html>). Although it is beyond the scope of this book, the FDA also released a number of regulations and guidance documents related to the use of software in developing, testing, and manufacturing drugs.

## The FDA Today (2010)

Today's FDA is best defined by its own mission statement:

### FDA's Mission Statement

The FDA is responsible for protecting the public health by assuring the safety, efficacy, and security of human and veterinary drugs, biological products, medical devices, our nation's food supply, cosmetics, and products that emit radiation. The FDA is also responsible for advancing the public health by helping to speed innovations that make medicines and foods more effective, safer, and more affordable; and helping the public get the accurate, science-based information they need to use medicines and foods to improve their health [5].

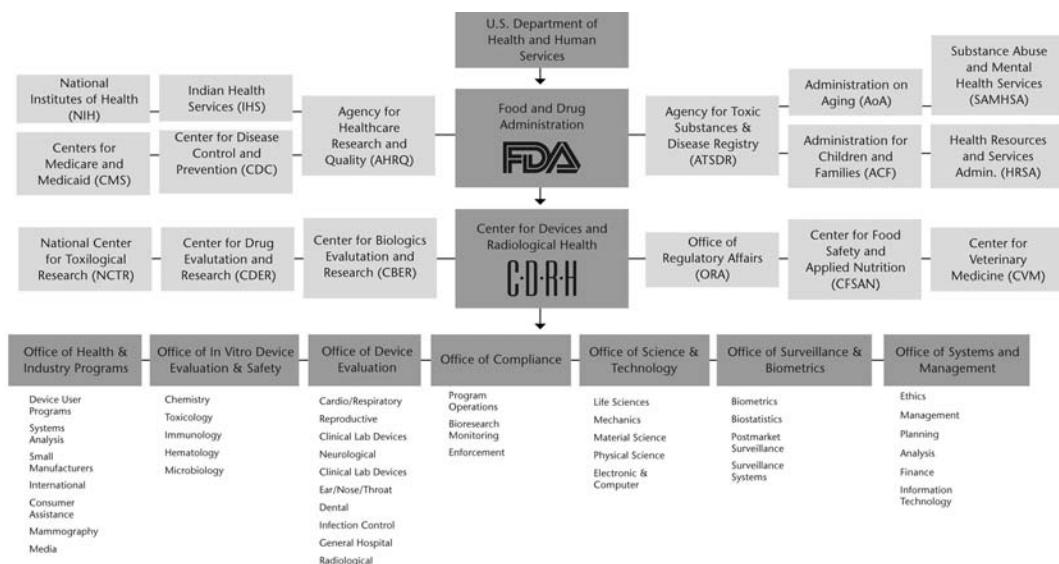
For our interest in medical device software in this text, let us focus on the first sentence. The FDA is responsible for "assuring the safety, efficacy, and security" of medical devices. To expand slightly on this, the FDA's responsibility is to assure:

- *Efficacy*: The device is effective in achieving the claims that are made for its intended and expected use;
- *Safety*: The device does not expose the patient, user, or other bystanders to undue risk from its use;
- *Security*: The device has been designed to protect patients, users, and bystanders from both unintentional and malicious misuse of the device.

This may not seem too difficult until you start to think about the magnitude of the task. The FDA today controls almost 25% of all consumer expenditures in the United States. The FDA is also responsible for regulating nearly a third of all imports into the United States. Admittedly, medical devices are only a small part of that responsibility.

The FDA achieves this with a staff of just over 8,000 employees (as of 2006). The FDA is one of 11 agencies or administrations within the Department of Health and Human Services. As shown in Figure 2.1 the FDA is comprised of seven "Centers." Of special interest to those in the medical device industry is the Center for Devices and Radiological Health (CDRH). CDRH, in turn is comprised of seven "Offices." The three Offices of most relevance to medical device development are:

1. *Office of Device Evaluation (ODE)*, the office to which premarket notifications are submitted for FDA approval prior to release to the market
2. *Office of Compliance*, which is responsible for enforcement of agency regulations. Device manufacturer investigations are coordinated through this office.



**Figure 2.1** Organizational chart for the FDA.

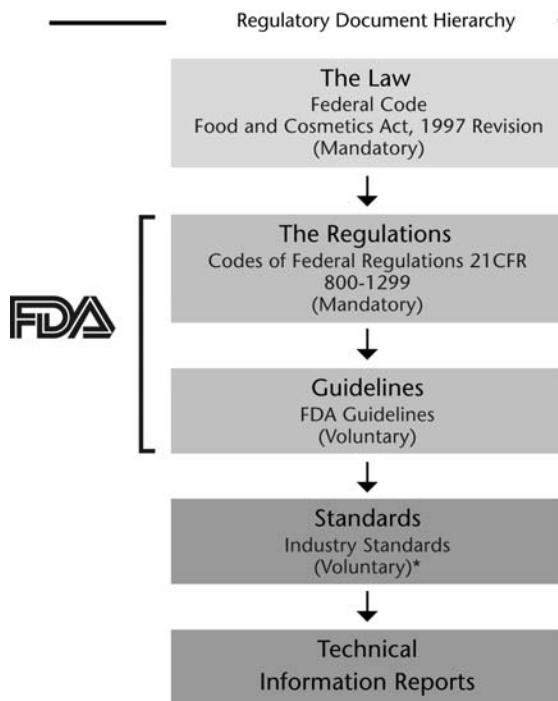
3. *Office of Science and Technology (OST)*, agency experts in this office specialize in narrow areas of science and knowledge. The FDA calls upon these experts for consultation and authorship of agency regulations and guidance documents. The experts may also be called upon to assist ODE and the Office of Compliance on the technically complex submissions agency investigations.

## How the FDA Assures Safety, Efficacy, and Security

Congress created and empowered the FDA for its mission in the Food Drug and Cosmetics Act. The agency further defined the regulations to meet its mission in Title 21 of the United States Code of Federal Regulations, which is usually referred to as 21 CFR. The hierarchical organization of the laws and regulations that empowered the FDA to control software validation is depicted in Figure 2.2. The regulations are broken into Parts. Parts 800 to 1299 pertain to medical devices and cover a wide range of responsibilities of the medical device manufacturer. Additionally, there are regulations in Part 11 that relate to electronic records and electronic signatures that define requirements for the validation of software that automates these functions.

Part 820 of the Code (usually referred to in shorthand as 21 CFR 820) was introduced in October of 1996 and contains the Quality System Regulations, which are often abbreviated as QSRs. These are available for free at <http://www.fda.gov/cdrh/fr1007ap.pdf> and are also located on the DVD that accompanies this text.

The QSRs are broken into Sections, which are usually identified to the right of the decimal point in the shorthand referencing. For example, two sections apply specifically to medical device software professionals, 21 CFR 820.30 and 21 CFR 820.70. 820.30 is often referred to as the Design Control Regulations, and they regulate the way that medical devices are planned, designed, developed, reviewed,



**Figure 2.2** Hierarchy of law and regulations that relate to software validation.

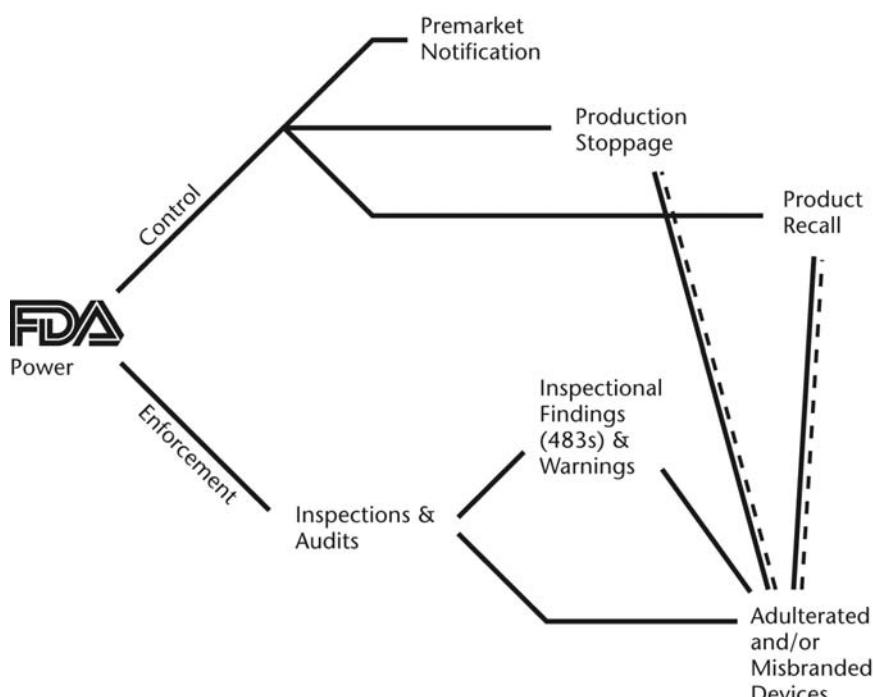
implemented, tested, and documented. This is the set of regulations that pertains most specifically to the topic of this text. 820.70 regulates Production and Process Controls. Subsection (i) specifically requires validation of any software used to automate any part of the production or quality system. As you will see in Part III, the 11 lines of regulation in 820.70(i) are very broad in scope and present special challenges of the software validation professional.

The laws and regulations are the tools that the FDA has to “assure safety, efficacy, and security,” but how exactly they enforce the regulations needs to be well understood by device developers because nearly everything a developer or validation engineer does is covered by regulation. Too often, developers in the industry tend to think of themselves as working in an “engineering cocoon” in which they know little about what is going on in the regulatory world outside. When they emerge from the cocoon they are often surprised by the number of regulatory requirements that were not met during the development process. That leads to significant energy and expense in reverse engineering and retro-documentation to satisfy the regulatory requirements. In general this is not pleasant work, and unfortunately it’s often not given the careful attention it deserves. The developers complain that the regulatory overhead is a waste of time, and it probably is when done this way. The intent of the Design Control Regulations is for developers to follow a well planned orderly development process with frequent review points to reduce the likelihood that devices will make it to the market with defects or that fail to meet the needs of the intended users. It is indeed no fun and of little value to retro-document a development project to make it look like it followed an orderly process when in fact it did not.

A power that the FDA and other international regulatory agencies that follow the same model have to enforce the regulations is summarized in Figure 2.3.

The same Medical Device Amendments empowered the FDA to require notification for agency approval prior to releasing a medical device marketplace (i.e., premarket notifications or PMNs). There are a variety of different types of premarket notifications, the details of which are beyond the scope of this discussion. The important point for now is that the FDA has the power to keep devices from being sold legally.

The Food Drug and Cosmetic Act granted in Section 704 the authority to inspect “any factory, warehouse, or establishment in which ... devices ... are manufactured, processed, packed or held, for introduction into interstate commerce or after such introductions.” This authority applies to files, papers, processes, controls, and facilities that may be used in a legal proceeding to prove that the “devices are adulterated or misbranded” or that the manufacturer is in violation of the Act. One can be certain that among the files and papers that are part of an inspection will be those that are specifically called out in the regulations. Of relevance to medical device software and the validation of software would be a Design History File (DHF), which is specifically required by 21 CFR 820.30(j). The DHF is the historical record of the design and development artifacts that should show the manufacturer was in compliance with the design control regulations during the development and validation of device. There is much to be said about what should and should not be in a DHF that is beyond the scope of this discussion. The important point here is that the FDA can inspect a manufacturer at any point in a medical device’s life cycle, and at that time may determine that the manufacturer is not in compliance. Agency



**Figure 2.3** Enforcement and control power of regulatory agencies.

response to such findings varies in severity depending upon the criticality of the inspectional findings and the manufacturer's willingness and ability to correct the issues.

This brings up a common misperception about the meaning of an FDA approval of a premarket notification (submission). It is common for device manufacturers (developers, in particular) to interpret FDA approval of a premarket submission as an acceptance of their quality system, and especially their design control processes. That is not the case. Approval of a premarket notification submission simply means that the agency was convinced from the evidence provided in the submission that the device is likely to be safe and effective, and that there is adequate reason to suspect that the manufacturer has a functioning quality system in place. This is not a substitute for the information gathered by inspectors during a quality system inspection. Many manufacturers are surprised when their inspections turn up a number of findings even though they were successful in getting approvals on a number of products submissions. FDA approval of a product submission is not synonymous with FDA approval of the manufacturer's quality system.

Section 519 of the Food Drug and Cosmetic Act granted authority to the FDA to require medical device manufacturers to submit reports to the agency for any significant problems reported from the field, or any suspicions of problems based on postmarket experience with the product. These reports are now known as Medical Device Reports (MDRs).

Since the Medical Device Amendments of 1976, the FDA has had the power to force manufacturers to recall ineffective, unsafe, or otherwise "adulterated" devices from the field. The same amendments gave the FDA the power to stop production of devices from manufacturers that are determined to be out of compliance with the regulations. The information the agency uses to come to the determination can come from the MDR's submitted by the manufacturers themselves or from information gathered during on-site inspections.

There are a number of other statutes and regulations relating to the corporate and individual penalties and controls that can be imposed on companies and their employees who are not in compliance with the law. These are not particularly relevant to this topic, but certainly topics which managers with approval authority and employees with direct contact with the FDA should be well aware.

In summary, the FDA has the power to keep a medical device that is suspected of being an ineffective or unsafe from ever getting on the market. That determination is made based on information provided by the manufacturer and a premarket notification submission. Additionally, they have the power to recall and/or stop production of unsafe or ineffective devices that do manage to find their way to market. That determination is made based on information provided by the manufacturer in Medical Device Reports, and by inspectional findings gathered by the FDA during on-site inspections.

## Quality System Regulations and Design Controls

By now you must be suspecting that the Quality System Regulations and their subset the Design Controls must be hundreds if not thousands of pages long. If so, you

would be pleasantly surprised to know that the QSRs are only seven pages long. The part of the QSRs referred to as the Design Controls is a little over a half-page long.

If the Design Control regulations are only a half-page long, what is all the fuss about? First of all, to clarify, the Quality System Regulation document is over 65 pages long, but the first 58 pages are the preamble that documents the rationale behind the regulations, and the agency's response to public comment prior to final release of the regulations. If you should decide to invest the time in reading the QSRs it would be well worth your time to also read the preamble. It adds a lot of insight into the intent behind the regulations.

The fuss seems to have to do with the following three perceptions:

1. There is too much detail in the Design Control QSRs;
2. There is too little detail;
3. There is a lot of folklore about detail in the regulations that really is not there.

The folklore category probably has a lot to do with the fact that people confuse the regulations, guidance documents, and their own corporate processes and procedures. As we examine the Design Control regulations, it will be apparent that there is not too much detail there. Those who complain that there is, probably actually belong in the folklore category, or have another agenda behind the complaint.

The Design Controls (21 CFR 820.30) are comprised of 10 subsections on the following topics:

- (a) General—Scope of Applicability
- (b) Design and Development Planning
- (c) Design Input
- (d) Design Output
- (e) Design Review
- (f) Design Verification
- (g) Design Validation
- (h) Design Transfer
- (i) Design Changes
- (j) Design History File

Each of these subsections is short. For example, subsection (g) on Design Validation says (in its entirety):

*(g) Design validation.* Each manufacturer shall establish and maintain procedures for validating the device design. Design validation shall be performed under defined operating conditions on initial production units, lots, or batches, or their equivalents. Design validation shall ensure that devices conform to defined user needs and intended uses and shall include testing of production units under actual or simulated use conditions. Design validation shall include software validation and risk analysis, where appropriate. The results of the design validation, including identifi-

cation of the design, method(s), the date, and the individual(s) performing the validation, shall be documented in the DHF.

That's it! That is the entire regulation having to do with validation. Note that software validation is just a mention in this regulation. The positive outlook to take from this is that the regulations do not contain hundreds of pages of minute detail on how devices are to be designed and developed. The mythological "regulatory overhead" is not rooted in regulation. The regulation outlines only high-level activities that few would argue are not good engineering practice. However, as is so often true, the devil is in the details when it comes to exactly what the terms mean, what activity and documentation artifacts will satisfy regulatory scrutiny, and how to conduct the activities in a way that increase your confidence in the safety and efficacy of the device.

## **Understanding How Regulation Relates to Getting the Job Done**

One might ask why the regulations are not more specific in itemizing expected documents to be produced, the format of those documents, and the details of the activities behind the production of the documentation. In the workshops and training sessions we conduct with medical device manufacturers all over the world, the most frequent request is, "Just show me what forms to fill in and I'll do it."

The objective of legislation and regulation should be to limit the details to only what is necessary to meet the intended objective. Why? There are several reasons:

1. Prior to adoption of the legislation or regulation, companies may already have put in place their own policies, processes, and procedures to accomplish the objectives of the regulation. Requiring, at the regulatory level or above, too much detail would require these companies to change what they already have in place, which may work just as well or better than what a detailed regulation would require. This would impose undue work and cost for these companies, and would run the risk of "breaking" a system that was working just fine before.
2. It would be difficult to prescribe a set of detailed activities and documentation that would apply in every conceivable situation from the present time into the future. To legislate or regulate at a very detailed level would result in portions of the industry struggling to fit into a regulatory system that does not quite fit them. That would create extra work and stress, for those companies and would result in no increase of safety or efficacy of the device.
3. Too much detail in the regulation would stifle creativity in finding better ways to control the design and development process, and would be slow to react to the evolving needs of the rapidly changing technologies upon which the design of many medical devices depend.

Great, too much detail in legislation and regulation is not a good thing, but that does not help the startup medical device company that does not already have a design control system (or more generally, a quality system) in place, or has a design

control system in place that also is written at a high level. Engineers and validation professionals in these companies face a very difficult, timely, and costly trial and error process of interpreting the regulatory intent unless they have worked with another medical device company before.

To aid these kinds of companies, or newcomers to the medical device industry, the FDA publishes a series of guidance documents to further explain their expectations of medical device companies for meeting the regulatory intent. The guidance documents are not legislative or regulatory requirements. They are explanatory or they recommend one way of accomplishing the goals of the regulation. Even if a medical device company chooses to comply with the regulation in a different way, it is valuable to be familiar with the guidance documents to understand the extent of compliance that may be expected. To be sure, there are problems with the guidance documents. There are overlaps in which two guidance documents may refer to the same regulation. There are contradictions between guidance documents. The documents are written at different points in time and are not synchronized. Both industry and regulatory thought on how to accomplish the intent of the legislation evolve with time. Two documents on related topics may be written 5 or 10 years apart and clearly show how the regulatory thinking has evolved.

Shouldn't the medical device industry and experts in medical device development and validation play some role in recommending "best practice" type guidance for the industry? The answer is clearly yes. We cannot expect regulators to be as experienced with product development as the device developers. The industry has created standards for this purpose. The standards typically are written by industry workgroups by consensus and compromise. They exist to satisfy those who desire to be told how to meet regulatory intent. These standards workgroups are typically organized by industry organizations, and frequently include members from the regulatory bodies. Compliance with standards also is voluntary.

Compliance with standards may afford some benefits to a device company during the premarket submission process. Certain standards are recognized by the regulating bodies. By adopting a recognized standard, a device company can be assured that the process defined by the recognized standard is acceptable to the regulatory agency that recognizes it. In certain submissions, that can eliminate the need to describe that process to the FDA. The submitting device company only needs to claim that they comply with the recognized standard. Once claimed, compliance companies do need to comply with the standards they claim they do, and that compliance will be checked during a regulatory inspection.

The FDA (actually CDRH) maintains a database of recognized standards at <http://www.accessdata.fda.gov/scrIpts/cdrh/cfdocs/cfStandards/search.cfm>. It is useful to check this when developing or revising one's quality system for available standards. Note that standards are routinely revised and updated too. This link will also show what version of a standard is the currently recognized version.

There is additional guidance available to the industry in the form of technical information reports (or TIRs). These are available, as are many standards, from the Association for the Advancement of Medical Instrumentation (AAMI) at their website [www.aami.org](http://www.aami.org). TIR's are different from standards in that they generally do not prescribe one particular way to accomplish a task. Instead, they are intended to be surveys of current practices in the industry applicable to a specific need.

## Medical Devices Sold Outside the United States

Most of the historical regulatory perspective above had to do with the United States FDA. For medical device manufacturers who intend to sell their devices outside the United States, there will be additional requirements for the regulatory agencies in the countries in which they sell their product. Luckily, there has been significant effort invested in harmonizing the regulatory requirements for medical devices on an international basis.

The European nations are good example of this. The European Parliament has agreed to a set of three directives known as the Medical Device Directives (MDDs). There are individual directives for active implantable devices, in vitro diagnostic devices, and all other medical devices. The principles relating to product development and device safety requirements that are agreed to in the Directives are transcribed into legislation in the member nations of the European Union. In this way the medical device regulations in each of the member nations are harmonized, and medical device manufacturers can satisfy the requirements of all nations of the Union simultaneously. These directives also are available online at no cost at [http://ec.europa.eu/enterprise/medical\\_devices/legislation\\_en.htm](http://ec.europa.eu/enterprise/medical_devices/legislation_en.htm) and are included in the content of the DVD that accompanies this book.

Each country in the European Union has an agency for administering the Medical Device Directives (MDDs) that is its equivalent of the United States FDA. These individual agencies are called Competent Authorities. Additionally, agencies known as Notified Bodies exist to assess how well medical device manufacturers comply with the MDD's requirements for the manufacturer's quality system.

There is an international standard (ISO 13485) entitled "Medical Devices—Quality Management Systems-Requirements for Regulatory Purposes." The objective of this standard is to "facilitate harmonized medical device regulatory requirements for quality management systems." A medical device manufacturer's quality system that is in compliance with the 13485 standard will also be in compliance with the requirements of the MDD. Notified Bodies certify the manufacturer's quality systems to be in compliance with the standard and also review the technical documentation relating to the development of the medical device to assess compliance with the requirements of the MDD. If the technical file and the quality system are both found to be in compliance, the manufacturer receives a CE Mark for the device in question, allowing it to be sold in the European Union. Manufacturers need only deal with one Notified Body to be able to market a device throughout the European Union. Individual nations within the Union may add additional requirements to those of the MDD. However, practically speaking, those differences have mostly to do with product labeling and reporting requirements.

There are many similarities between the organization of 13485 and the ISO-9001 quality standard for manufacturers. ISO-13485 has regulatory compliance requirements that are inappropriate for the more general ISO-9001 standard. ISO-13485 is more detailed in its requirements for documentation, planning, and process-driven development than ISO-9001 is and somewhat less prescriptive than the FDA's Quality System Regulations.

Although obviously not part of the European Union, Canada requires compliance with 13485 of any medical device manufacturer selling medical devices in Can-

ada. Asian nations have their own regulations related to medical devices. The complexities of international medical device regulation deserve a textbook of their own and will not be developed further here. Thankfully, efforts have been underway for some time to develop true worldwide international harmonization of medical device regulations. The official workgroup responsible for this is called the Global Harmonization Task Force (GHTF). The history, progress, and output of this workgroup can be monitored at their website at [www.ghtf.org](http://www.ghtf.org).

## References

- [1] “Milestones in U.S. Food and Drug Law History,” *FDA Backgrounder*, May 3, 1999 (Updated August 5, 2002). Available at [www.fda.gov/opacom/backgrounder/miles.html](http://www.fda.gov/opacom/backgrounder/miles.html).
- [2] Munsey, R. R., “Trends and Events in FDA Regulation of Medical Devices Over the Last Fifty Years,” *Food and Drug Law Journal*, Vol.50, Special Issue (1995), pp. 163–77. Available at [www.fdl.org/pubs/Journal%20Online/50th%20Anniv/art15.pdf](http://www.fdl.org/pubs/Journal%20Online/50th%20Anniv/art15.pdf).
- [3] Swann, J.P. “History of the FDA,” FDA History Office. Adapted from *A Historical Guide to the U.S. Government*, George Kurian (ed.), New York: Oxford University Press, 1998). Available at [www.fda.gov/oc/history/historyoffda/default.htm](http://www.fda.gov/oc/history/historyoffda/default.htm).
- [4] Leveson, N., and C. S. Turner, “An Investigation of the Therac-25 Accidents” (reprinted from IEEE Computer, Vol. 26, No. 7, 1993, pp. 18–41). Available at [http://courses.cs.vt.edu/~cs3604/lib/THERAC\\_25/Therac\\_1.html](http://courses.cs.vt.edu/~cs3604/lib/THERAC_25/Therac_1.html).
- [5] <http://www.fda.gov/opacom/morechoices/mission.html>.



# The FDA Software Validation Regulations and Why You Should Validate Software Anyway

In this chapter, the Therac 25 incident will be covered in more detail to gain an understanding of the state of the industry before the FDA began requiring software validation. This should make it clear why the regulations were necessary. The relevant regulations and guidance documents themselves are examined to explain how they fulfill the regulatory need for requiring validation. Finally, a short section is presented on the advantages of validation for the medical device manufacturer.

## Why the FDA Believes Software Should Be Validated

In the preceding chapter, the early history of the FDA's role in medical device software was outlined. The Therac 25 disaster certainly played a role in the government becoming more proactive in regulating medical device software.

To repeat from the preceding chapter, the FDA's interest in medical device software, as well as quality system software starts with its mission statement: To assure the safety, efficacy, and security of medical devices of which software is a component. The same applies to software which has played a role in the design, development, production, or control of quality of the device.

Later in this chapter, the regulatory roots of software validation are traced. Software validation is mentioned specifically in the regulation. Interestingly, electronics, mechanics, hydraulics, radiation, and the theory and science that are at the foundation of the medical device are not specifically called out. They are presumably intended under device verification and device validation, but they are not specifically mentioned the way software is. Is it because software is different?

Well, in short, yes. Unlike electronics or mechanical assemblies, software does not corrode, wear out, or have statistical failures of subcomponents. That's good. However what that means is that, with few exceptions, all software failures are really design or development failures. The exceptions generally are related simply to loading the wrong software into the device during production. It's generally accepted that the software in most medical devices is more complex than the electronics. That includes the processors that execute the software, and the remainder of the device electronics, sensors, and effectors that the software controls. It is because of this complexity, except for the very simplest examples, that software cannot be fully tested for every combination of potential pathway through the code, or for every historical pathway and accumulation of data.

Software is easily changed. Changes may not be obvious until they are actually used, whereas a change to a circuit board, keypad, or other tangible element of the device may be very obvious. Anyone who has ever tried to write any software is very aware of the fact that any small change to the software can have totally unintended consequences. Although this is also true to some extent to the hard elements of the device, the effects of change to software seem to be more prone to deleteriously affecting software for unrelated functionality.

So, if almost all software defects are created during design and development, then certainly the regulatory agency should focus their attention on the design and development activities to reduce the number of unsafe defects related to software.

## Therac 25

The Therac 25 incident was much studied at the time and yielded a wealth of possible mechanisms for unsafe software defects that could be embedded in medical devices. One of the best summaries of the studies was written by Nancy Leveson [1], where she itemized the following factors that led to the Therac disaster.

*Overconfidence in software.* Software was not considered in the hazard analysis for the device, hardware was suspected initially for the failures, and many of the safety features were implemented in software without adequate backup mechanisms in case the software failed.

*Confusing reliability with safety.* Although software may operate correctly 99.999% of the time (making it very reliable), the particular set of inputs or sequences of events that lead to erroneous behavior will lead to that erroneous and potentially unsafe behavior 100% of the time. Such a system is unsafe, but highly reliable. The high reliability led to initial suspicions that the problems were due to causes other than software.

*Lack of defensive design.* The device was not designed with adequate cross checks, error detection, and error-handling capabilities. No service or performance logs were implemented that could assist in tracing the problems to their root cause quickly.

*Failure to eliminate the root causes.* The Leveson analysis contains numerous examples of this. In one of the more blatant examples, pressing the P key (to proceed) on the keyboard during one of the software's failure modes could result in an unwanted and unreported dosage of radiation. After five such "proceeds," the system would shut down. The manufacturer's response to this was to reduce the number of "proceeds," resulting in shutdown from five to three to limit the amount of overdosage attacking the symptom but not the root cause. The sequence of events leading to this error state involved the operator's use of the up arrow key when positioning the cursor for editing data on the screen. The response of the manufacturer, Atomic Energy of Canada Limited (AECL), was to send out a letter to users of the Therac system requiring that "the key must be removed and the switch contacts

fixed in the open position with electrical tape or other insulating material.” One can only imagine how desperate the company must have been to offer some help to the users as quickly as possible. This is, however, a classic example of treating the symptom of a problem rather than addressing its root cause.

*Complacency.* Because prior generations of the device had been safe and reliable, the manufacturer was somewhat complacent in responding to initial reports of the problem. Initial reports were attributed to operator error. The complacency actually spread to the users themselves as they became accustomed to frequent error messages and shutdowns of the system.

*Unrealistic risk assessments.* As mentioned before, software was not even considered as part of the hazard analysis for the system. Had it been, it may have been subject to some of the other unrealistic risk assessments that were applied to the system hardware. At one point a switch was replaced and the manufacturer estimated that the likelihood of failure of the system had been improved by five orders of magnitude.

*Inadequate investigations or follow-up on accident reports.* Perhaps related to complacency, or experience with prior generations of the device, the manufacturer was slow to respond to reports from the field and inadequately investigated the user claims when they finally did respond.

*Inadequate software engineering practices.* Much of Leveson’s analysis was focused on software engineering practices. Problems cited in Leveson’s work include:

- Software requirements and design documentation were written as an after-thought.
- There were no quality systems in place to control the design and development of software.
- Software design may have been overly complicated.
- Only system-level software testing was performed. No unit level or integration level testing was evident. There was no plan for regression testing of all software changes.
- The software’s user interface was confusing to the user. Error messages were cryptic. The user manual was difficult to understand, provided no information to explain the meaning of error codes, nor did it provide any guidance for the user when error codes were displayed.

*Software reuse.* Although it seems counterintuitive that reuse of software from prior generations of the device could lead to defects in unsafe conditions of a next-generation, that was exactly the case with Therac. Limited testing was done with reused software components, resulting in defects in the new device. In at least one case this was related to an assumption made by the author of the original software module that was not met in the next-generation device. One can presume that the engineers were complacent about testing this module because it had worked

flawlessly on prior generations of the device. One can also assume that the documentation for the reused module was on the “lean” side, and that the assumptions were not documented.

*Safe versus friendly user interfaces.* The user interface had been modified to speed data input by the operator of the device. Programmers assumed that the operator would check the values in the data fields carried over from prior entries before submitting them to the device for execution. That was not the case, and erroneous data entry lead to unsafe dosage levels for the patients. The user interface was clumsy. The users were inconvenienced and deserved more efficient data entry process. However the user interface as implemented did not consider the impact it might have on safety. It was not designed to reduce the probability of user error, nor did it check for user error after the data was entered.

Up to this point in time, the FDA had had little experience in dealing with software-related medical device failures of this severity or at this scale. It was recognized that regulations and guidance had to be provided to help prevent similar situations in the future. The resulting guidelines from the FDA relating to medical device software addressed the factors that caused the Therac disaster.

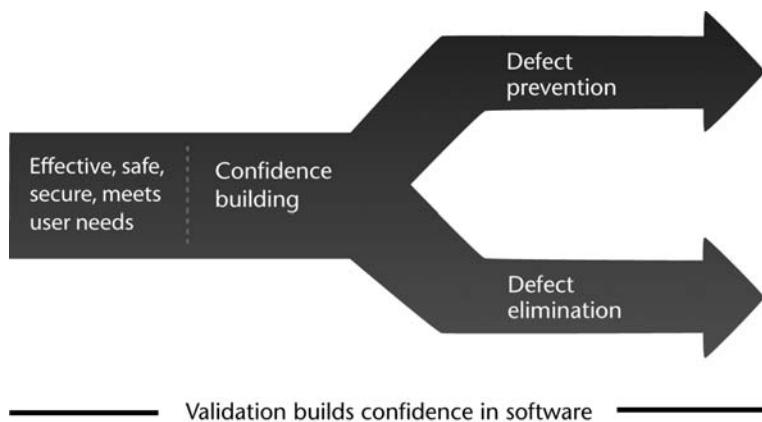
The details of what may have caused the Therac 25 disaster are important for the following reason. They provide a good historical basis for how seemingly minor oversights can result in severe consequences for users or patients. It useful to refer back to Therac 25 from time to time to be reminded why we do what we do related to software validation, and why the FDA wants a role in regulating design, development, and validation.

It is also disturbing to look back the causal factors identified in the Therac investigation and realize that the industry is still struggling with many of the same issues today, over 25 years later. There is virtually nothing in that list of factors that we can say that we as an industry have totally eliminated with today’s advanced technologies and techniques and another 25 years of experience behind us!

## Building Confidence

Software validation in very simple terms is simply building confidence in the software before it is released for widespread use. That confidence should be established in the reliability and effectiveness of the software to complete its specified tasks. Additionally, confidence in the safe operation of the device controlled by software should also be established. One wants to be confident in the safety of a device when it is used and working as anticipated, but one also wants it to operate safely when it is not used as anticipated and when hardware or software subsystems fail.

Let’s consider how one builds confidence in something like software that, admittedly, cannot be fully tested, and whose defects are created at design time. A two-prong approach as shown in Figure 3.1 is needed that (1) prevents defects from being implemented in the first place, and (2) finds and eliminates those defects that do manage to manifest themselves in the device.



**Figure 3.1** The two prongs of building confidence in software.

There is some ongoing confusion and debate in the industry about what exactly software validation means. Years ago there was widespread understanding and belief that software validation meant testing. (That is still true among many startups, small companies, and even established large companies.) When the General Principles of Software Validation (GPSV) was released it was clear that the FDA had much more in mind than just testing. The GPSV sometimes draws comments from the industry that it is too broad in scope, covering many good engineering practices related to software development, in addition to software testing. However, if one understands that validation is about building confidence in the software, and that the two prongs of confidence building include prevention as well as elimination, one can see why the FDA concept of software validation extends into software development activities.

## The Validation Regulations

With the Therac experience behind them, the FDA regulators had adequate incentive to begin articulating and regulating by requiring (or at least strongly suggesting) their vision of software validation. The FDA had a number of precedents upon which to rely when they authored the Quality System Regulations relating to software validation. The terminology (verification and validation) selected in the regulation continues to be a source of confusion in the industry despite the amount of guidance the agency has offered on this point. Part of the confusion comes from the fact that other industries and standards use the terms verification and validation, have different meanings of these terms. The IEEE and ISO standards that were available at the time the QSRs were created were clearly influential on the FDA's understanding and ultimate parsing of the definitions of software verification and validation in the regulation. Consequently, professionals from other industries have come to the medical device industry with a different understanding of the basic terminology that underlies all of the processes and activities upon which verification and validation rely.

In the FDA's preamble to the Quality System Regulations, it is clear that the FDA and the medical device industry struggled with the definition of validation. From the QSR preamble:

... FDA has adopted the ISO 8402:1994 definition of validation. "Validation" is a step beyond verification to ensure the user needs and intended uses can be fulfilled on a consistent basis.

The definitions of verification and validation in the definitions section of the QSRs was finalized as follows:

820.3 (z) Validation means confirmation by examination and provision of objective evidence that the particular requirements for a specific intended use can be consistently fulfilled.

(1) Process validation means establishing by objective evidence that a process consistently produces a result or product meeting its predetermined specifications.

(2) Design validation means establishing by objective evidence that device specifications conform with user needs and intended use(s).

(aa) Verification means confirmation by examination and provision of objective evidence that specified requirements have been fulfilled.

The specific meaning of these definitions as they relate to medical device software and quality system software will be examined in great detail later in this text. However, at this point it is interesting to note in the above quoted section of the QSRs the hierarchical structure of these definitions. Validation is comprised of process validation and design validation. Process validation is more related to production processes. Design validation is considered as separate from process validation, and verification is a subset of validation. The FDA has broken this down to a much more detailed extent in the GPSV.

The above-quoted section came from the definition of terms used in the regulation. The actual regulations related to design verification and validation, from which software validation took root are the following (from Section 820.30, the Design Controls):

(f) Design verification. Each manufacturer shall establish and maintain procedures for verifying the device design. Design verification shall confirm that the design output meets the design input requirements. The results of the design verification, including identification of the design, method(s), the date, and the individual(s) performing the verification, shall be documented in the DHF.

(g) Design validation. Each manufacturer shall establish and maintain procedures for validating the device design. Design validation shall be performed under defined operating conditions on initial production units, lots, or batches, or their equivalents. Design validation shall ensure that devices conform to defined user needs and intended uses and shall include testing of production units under actual or simulated use conditions. Design validation shall include software validation and risk analysis,

where appropriate. The results of the design validation, including identification of the design, method(s), the date, and the individual(s) performing the validation, shall be documented in the DHF.

Some parts of the above design control regulations are strictly procedural to establish the “objective evidence” for verification and validation to allow for subsequent assessment by corporate management and by the Agency. Unfortunately, the design inputs and design outputs mentioned in the design verification regulation have led to their own history of misunderstanding and debate in the industry. The FDA has given some guidance on this terminology in the Design Control Guidance for Medical Device Manufacturers (1997). Detailed discussion and clarification of this is deferred until Chapter 6.

The regulation for design validation specifically includes software validation (unfortunately with no definition of exactly what that means) and does clarify that validation is to be performed on both initial production units, and on production units under actual operating conditions. The regulation is clear that validation is primarily focused on ensuring that the devices meet user needs: safely, effectively, and consistently.

We tend to focus our attention for software validation on the software that is embedded in the medical device. That is probably well justified since software that is embedded in the device, or that is considered a device in its own right, can directly cause harm to the user or patients.

Other kinds of software can indirectly cause harm to users or patients. Software that automates device design, development, production, or other parts of the manufacturer’s quality system such as complaint handling and CAPA can indirectly result in harm to a patient by causing or allowing defects in the device to get through the quality system. Consequently, embedded device software is not the only software over which the FDA has regulatory oversight. Since the mid-1980s, the FDA published guidance documents related to software used in the production of drugs. This has carried over to the device regulations and currently there are two regulations that control software that indirectly can result in harm: 21 CFR 820.70 (i) and Part 11.

The first of these regulations is 21 CFR 820.70(i), which regulates automated processes. That entire regulation is as follows:

- (i) Automated processes. When computers or automated data processing systems are used as part of production or the quality system, the manufacturer shall validate computer software for its intended use according to an established protocol. All software changes shall be validated before approval and issuance. These validation activities and results shall be documented.

That’s it. The third part of this textbook is devoted entirely to the validation of this type of software, which might be called quality system software or nondevice software, quality and production system software, or automated process software for quality systems, or some combination of any of these. This is not a small topic, nor is it a simple topic, nor is it less important than the embedded device software validation. Some of the techniques that are used for validating embedded device software are applicable in the validation of nondevice software. However, valida-

tion of software that is acquired for use off-the-shelf is much different from validation of software that is custom developed. In the context of this chapter those differences will not be discussed, but will be discussed in great detail in Part III.

The second of the regulations requiring validation of software that could result indirectly in harm to a patient is the Part 11 (eleven) regulation that covers electronic records and electronic signatures (commonly referred to as ERES). This is a somewhat longer regulation that will not be quoted here, but is included in the accompanying DVD. The Part 11 regulation as it currently exists sets out requirements for ERES software. The regulation also has a requirement for validation of ERES software. The scope of this regulation has had a long history of debate and confusion in the industry. The most fundamental issues have to do with the definition of ERES software, and to which records and signatures the Part 11 regulation applies. Since this is not a text about Part 11 we will leave it to others to sort out those issues. We will focus on the validation of ERES software in Part III, which actually will be treated almost identically to that of other nondevice software.

Part 11 has been explained and reexplained by the FDA since its initial release. As of this writing, the FDA has provided the industry with guidance on its enforcement intent related to this regulation. That guidance [2] is of more limited scope than one might interpret from reading the regulation alone. Part 11 is currently being rewritten by the FDA but its expected release date is unknown as of this writing.

Let's leave this discussion for now with an understanding of why the FDA wants the industry to validate software, and an initial high level understanding of the components the FDA suggests are part of software validation.

## Why You Should Validate Software Anyway

Now that you have some understanding of why the FDA thinks software validation is so important for medical devices, let us take a look at whether validation is a good business practice as well. If software validation is good for the FDA (i.e., if the FDA thinks it's good for the device), then why shouldn't it be good for you and your company? Well, it is.

Even if validation were nothing more than testing, there certainly would be business benefits even from a systematic testing program. Testing early and often would allow developers to identify defects early in the development life cycle before they get too deeply embedded and intertwined with other functionality, documentation, and test procedures. Numerous studies have shown that the cost of correcting defects is as much as 100 times less expensive early in the development life cycle than it is late in the development life cycle. The costs probably greatly exceed the hundred to one ratio if one takes into account the cost of recalls and trying to diagnose and repair defects in the field (not to mention the potential cost of human harm or even death). Even if the criticality of field defects is not at the level of recall or human harm, there is a cost involved in service, maintenance, and to the company's reputation even for nuisance defects. It should be clear that testing to eliminate as many defects as possible, as early as possible, will result in a net savings to the manufacturer.

Validation, as discussed in earlier sections of this chapter, is as much about defect prevention as it is about defect detection and elimination. The benefits of a well-documented and reviewed requirements development process are many. Formal identification of clinical, user, patient, and market needs is key to the development of a successful product on the market. The traceability of those needs through device specifications and ultimately to the tests that verify the correct implementation of those specifications assures the manufacturer that the device is correctly implemented to meet specifications, and, is in fact the device that the clinicians, users, patients, and market actually need and want. Doesn't it make good business sense to develop the product that your customers actually need?

As we will see later chapters, some of the documentation that the FDA suggests also has great benefits to the business. It makes sense that software developers who develop software to fulfill a fixed number of requirements in specifications will do that more efficiently than if they have to discover the requirements as they write the software. It makes sense that larger development teams will be better able to work together efficiently if they do so through documented requirements specifications, and design specifications. It makes sense that test teams can more efficiently and thoroughly test the software and the device itself if they do so through well documented requirements specifications, and usage models.

Of course, it always makes good business sense to comply with regulatory requirements. One doesn't need an MBA to appreciate the financial benefits to a company if its premarket submissions are not delayed due to quantity or quality of information provided in the premarket submission. It is also easily appreciated that the company benefits by reducing the number of complaints, medical device reports (MDRs), investigational findings, and other regulatory enforcement actions through good engineering practices that are part of the FDA's vision of software validation.

The developers themselves actually benefit from a systematic software validation program. Applying metrics to the validation program gives developers some measure for better estimating project costs and schedules. Similarly, defining metrics for the rate of defect identification give some measure of the confidence in the software and its readiness for release for human use. Critical review of requirements and designs is really a debugging activity for finding defects before the engineering effort is expended in implementing defective requirements or designs. This saves time, money, and frustration for the development group. Keeping the entire development team on track by communicating through reviewed documentation eliminates expensive rework in the course of a development project.

In summary, the FDA has adequate evidence that inadequate software validation can result in unsafe or ineffective devices. Software validation is required by law through the Quality System Regulations. The General Principles of Software Validation (GPSV) are defined by the FDA in a guidance document. Regulatory wisdom and good business common sense tells us that validation is good for the device and good for the company. A good validation program can actually reduce the time-to-market for a medical device. So let's stop fearing and resisting software validation and find out what it really is all about.

## References

- [1] Leveson, N. G., *Safeware, System Safety and Computers*, Addison-Wesley Publishing Company, 1995.
- [2] “Guidance for Industry. Part 11, Electronic Records; Electronic Signatures—Scope and Application,” U.S. Department of Health and Human Services, Food and Drug Administration, August 2003.

# Organizational Considerations for Software Validation

Recall the two-prong approach to software validation described in the previous chapter: defect prevention and defect elimination. A good quality system and a corporate organization that supports the quality system are the key elements of a defect prevention program. It has been mentioned before that the FDA's concept of software validation will make it clear that software validation involves much more than software testing. There is a large degree of overlap of validation activities and development activities because of the defect prevention component of software validation. The quality system and organization of the medical device manufacturer affect the quality of medical device software at least as much as the skills of the software developers and testing engineers. Having a formal design control process with the right balance of detail and flexibility is the best way an organization can evolve toward improving the quality of software, and reducing the time-to-market for that software. Yes, that is not a mistake: a well structured design control quality system can actually reduce time-to-market while improving the quality of the software.

## Regulatory Basis of Organizational Responsibility

The executive management of a medical device manufacturer, and indeed the manufacturer of any type of product, can impact the quality of the device, and specifically the quality of the software that is embedded in the device, by carefully structuring the organization of the company in close coordination with the design of the quality system.

There is a regulatory basis for this executive responsibility in the FDA QSRs. In definitions section 820.3 (n), the FDA defines who management with executive responsibility is. This is not a definition based on title within an organization, nor does the FDA prescribe a specific organizational structure. The requirement is based on responsibility. The FDA's definition of executive management, or more precisely management with executive responsibility, is:

820.3 (n) Management with executive responsibility means those senior employees of a manufacturer who have the authority to establish or make changes to the manufacturer's quality policy and quality system.

This definition does not say that executive management is only a company CEO, or senior vice presidents, or any other specific title of the organization. What it does say is that management with executive responsibility includes any title in the organization that is authorized to create or maintain the device manufacturer's quality sys-

tem. A key component of any quality system is the definition of responsibility within that system for the creation and maintenance of the quality system, and for any of the activities that are specifically required by the system. The regulatory requirements for management responsibility again are from the FDA's QSRs:

820.20 Management responsibility.

(a) Quality policy. Management with executive responsibility shall establish its policy and objectives for, and commitment to, quality. Management with executive responsibility shall ensure that the quality policy is understood, implemented, and maintained at all levels of the organization.

(b) Organization. Each manufacturer shall establish and maintain an adequate organizational structure to ensure that devices are designed and produced in accordance with the requirements of this part.

(1) Responsibility and authority. Each manufacturer shall establish the appropriate responsibility, authority, and interrelation of all personnel who manage, perform, and assess work affecting quality, and provide the independence and authority necessary to perform these tasks.

(2) Resources. Each manufacturer shall provide adequate resources, including the assignment of trained personnel, for management, performance of work, and assessment activities, including internal quality audits, to meet the requirements of this part.

(3) Management representative Management with executive responsibility shall appoint, and document such appointment of, a member of management who, irrespective of other responsibilities, shall have established authority over and responsibility for:

(i) Ensuring that quality system requirements are effectively established and effectively maintained in accordance with this part; and

(ii) Reporting on the performance of the quality system to management with executive responsibility for review.

(c) Management review. Management with executive responsibility shall review the suitability and effectiveness of the quality system at defined intervals and with sufficient frequency according to established procedures to ensure that the quality system satisfies the requirements of this part and the manufacturer's established quality policy and objectives. The dates and results of quality system reviews shall be documented.

(d) Quality planning. Each manufacturer shall establish a quality plan which defines the quality practices, resources, and activities relevant to devices that are designed and manufactured. The manufacturer shall establish how the requirements for quality will be met.

(e) Quality system procedures. Each manufacturer shall establish quality system procedures and instructions. An outline of the structure of the documentation used in the quality system shall be established where appropriate.

To paraphrase, a device manufacturer needs a quality system with:

1. An organizational structure that supports compliance with the quality system;
2. Procedures that define that system;
3. Defined responsibilities within the organization;
4. Routine review and improvement of that system.

That's it. That is the sum total of regulation related to the organization of a medical device manufacturer.

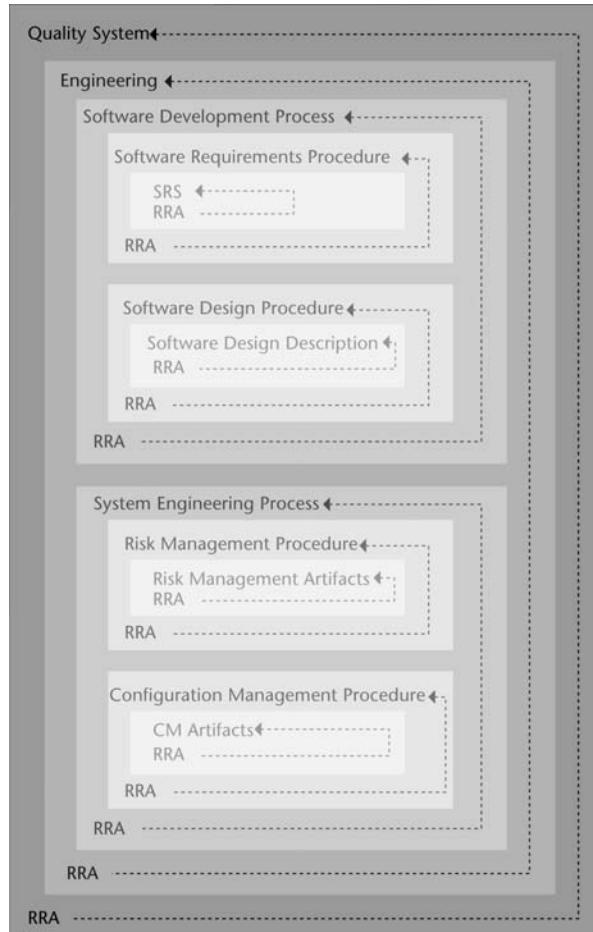
## A Model for Quality Systems

A manufacturer's quality system covers all aspects of the business related to the design, development, production, and control of quality of the device. For the purpose of this text, the focus will be on the design control regulations since they are the ones that are most relevant to software validation.

The diagram in Figure 4.1 is an abbreviated representation of part of a quality system related to the design and development of medical device software. Due to limitations of page space, the specific processes and procedures represented in this figure were chosen only to make the following points:

1. The quality system model as depicted in Figure 4.1 has a structure that mimics the structure of the organization itself. The responsibilities for the quality system parallel the responsibilities of the device manufacturer's organization. That is, responsibility for quality lies within the various disciplines within an organization, not centralized in one large quality organization.
2. The quality system is a system. It is a hierarchically organized system of policies, processes, procedures, plans, and other lower-level supporting guidelines. (These terms are the terms used in this text. Organizations use a number other terminologies to define the architecture of a quality system.) It is not intended to be set of rules that are written once, and written perfectly the first time, then ignored and forgotten.
3. The quality system is self-regulating, and improves as it evolves over time. The "review, revise, and approve (RRA)" requirements of the system are the key elements for the regulation and evolutionary improvement of the system.
4. The quality system (at least the part that is related to product design and development) includes requirements for review, revision and approval of the specific documents to be created as part of a product development program as well as requirements for the review, revision, and approval of the processes and procedures that make up the quality system itself.

The review, revise, and approve (RR&A) activity is extremely important for the regulation and evolution of the quality system. Careful consideration is necessary when designing the RR&A requirements for each process and procedure as well as



**Figure 4.1** Process model of a quality system.

for each output of the quality system, which includes all design and development artifacts. The RR&A requirements are primary contributors to quality systems that impede product development time to market. Consequently, they represent a primary opportunity for streamlining design control quality systems that are viewed as overly burdensome, slow, and which few in the organization respect or use. Recommendations for RR&A requirements will be covered in more detail at the end of this chapter.

### Roles, Responsibilities and Goals for the Quality System

Figure 4.1 diagrams a process for designing and maintaining a quality system used for developing and validating medical device software. As mentioned above it is critically important to understand the roles and responsibilities within that quality system. Structuring the quality system in a way that mirrors the organization of the company helps to clarify the responsibilities for defining and improving the quality system.

It is useful to itemize the goals and expectations of your quality system. The following goals are those that we have developed for ourselves, and that we recommend for our clients who come to us for advice in developing or improving their quality systems. For the purposes of this text, the goals are restricted to those that are relevant to software development and validation.

1. *Consistency of quality.* Not only should a quality system help to produce high-quality outputs, we want consistent quality regardless of the product, project, developer, manager, or tester. The quality system needs to be detailed enough to provide consistency.
2. *Assurance of compliance.* The quality system needs to be designed with an awareness of the regulatory requirements for medical device manufacturers. Project teams whose activities are guided by the quality system should rely upon the quality system for compliance with regulatory requirements. That is, users of the quality system should not need to refer constantly to regulation or regulatory guidance documents.
3. *Corporate memory.* There needs to be a place in the quality system for enough detail in procedures to repeat past successes and to avoid past mistakes. The quality system can become an element of the corporate memory that evolves over time and is passed on to succeeding generations of product developers and testers.
4. *Flexibility.* The quality system needs to be detailed enough to assure the first three goals, yet flexible enough to allow the discipline experts (that is, subject matter experts) to exercise their expertise, and find ways to improve upon the quality system.

### The Structure of the Quality System

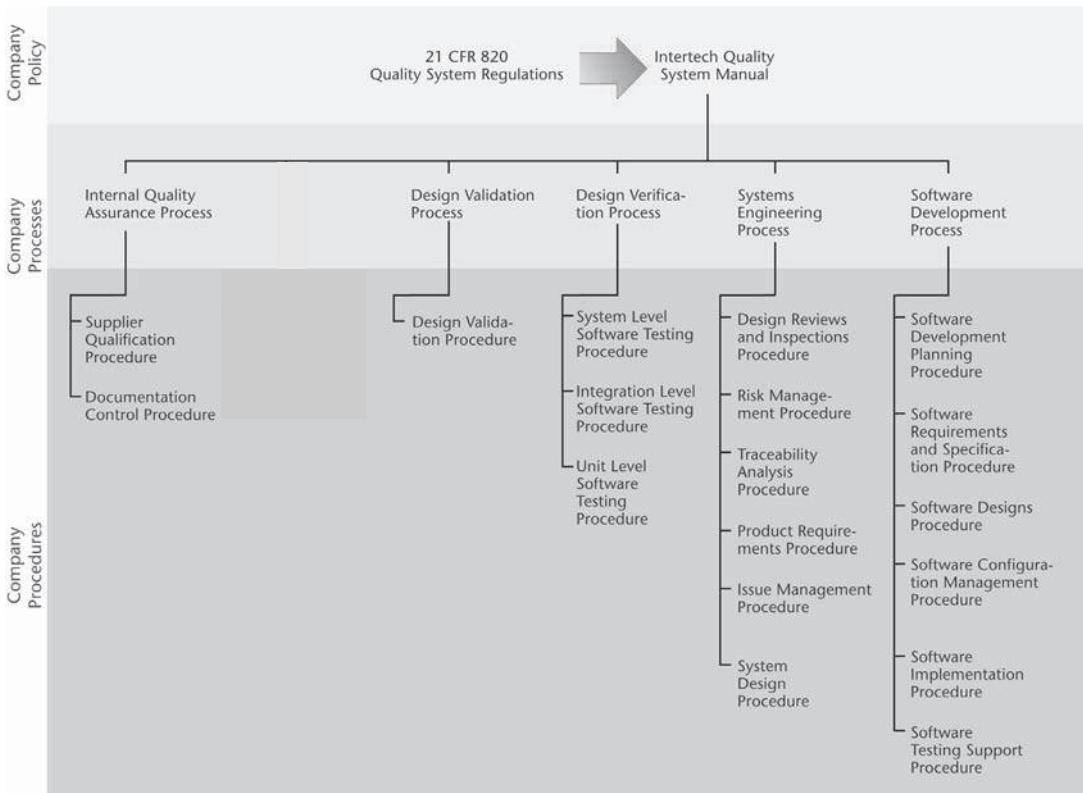
It is useful to represent the structure of a quality system in the form of a diagram for communicating with coworkers. There is some regulatory basis for this in 21 CFR 820.20 (e):

An outline of the structure of the documentation used in the quality system shall be established where appropriate.

It can be done in outline form, in text, or in diagrammatic form as we have chosen to do in Figure 4.2.

Once the structure of the quality system has been determined, the terminology must be defined to describe that structure. The terminology used in this text is somewhat different from terminology used by other standards and organizations. The terminology we will use starts with the four Ps:

1. *Policy:* Corporate quality requirements to fulfill regulatory requirements and recommendations. Policy is a declaration of which regulations, guidances, and standards with which the company claims to comply.
2. *Process:* The activities required for a given corporate discipline, and the requirements for those activities needed to fulfill claims in the policy.



**Figure 4.2** Swim lane diagram of a simple quality system.

3. **Procedure:** The detailed instructions for carrying out a given activity in a way that complies with the requirements for that activity in the process. Note that the quality system suggested here can support multiple procedures for the same process activity.
4. **Plan:** A project-centric strategy for meeting the needs of the project while staying compliant with the requirements of the process. A plan identifies which procedures will be followed for a project, and identifies gaps where procedures may need to be developed or modified for the project.

### Quality System Processes

The vertical columns or “swim lanes” in Figure 4.2 represent disciplines within the company. Since the subject of this text is software validation, only the disciplines that are relevant to software development and software validation are included. Each discipline is responsible for its own quality system process that defines how that discipline executes its responsibilities within the confines of the larger quality system and external regulatory requirements. As shown in Figure 4.2, the disciplines we have defined that are relevant to software development and validation are:

- *Software development*: development activities;
- *Software verification and validation testing*: testing activities (note “validation” activities are shared throughout the discipline processes);
- *Systems engineering*: traditional systems engineering functions and many of the cross functional activities associated with software development and validation;
- *Design validation*: activities that assure that the device that is being developed is the device needed by clinicians and patients;
- *Quality assurance*: activities that monitor our own compliance to the quality system.

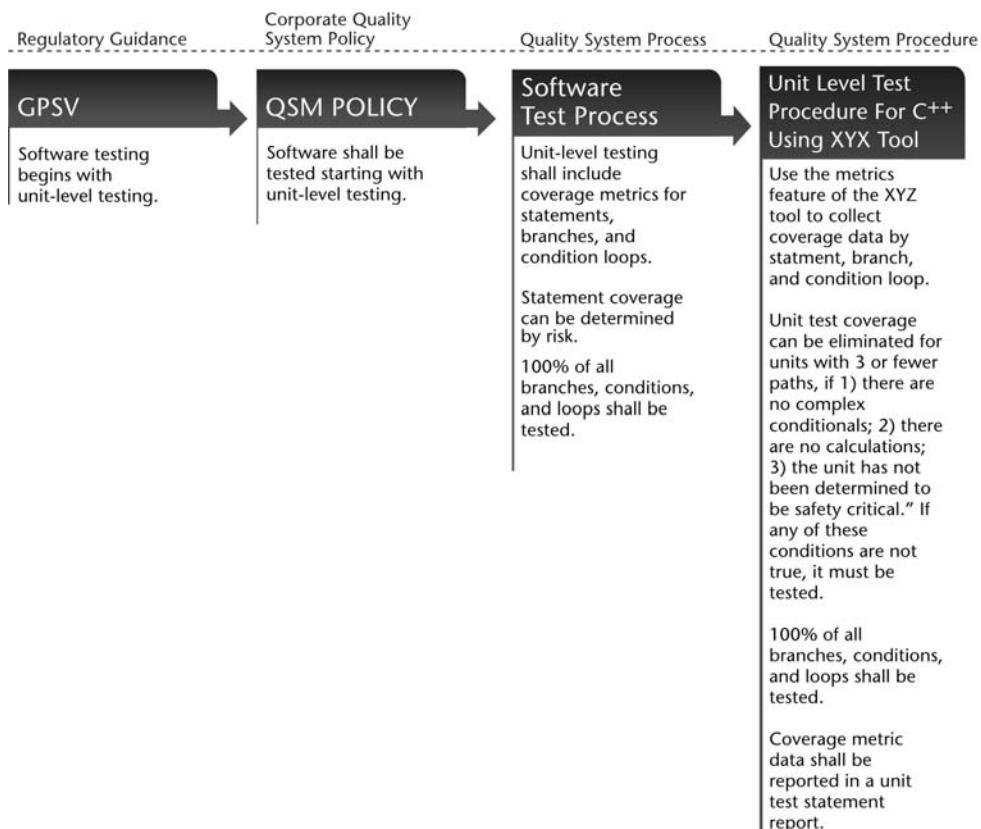
Each of these company processes defines the activity requirements for that particular discipline in the company. The discipline manager is the person responsible for the creation and maintenance of the discipline’s process. These requirements originate in the company’s quality policy, which is also sometimes referred to as the quality system manual. The quality policy requirements are anchored in regulatory requirements (as shown in the figure) and recommendations from regulatory guidance documents as well as from industry standards to which the company has voluntarily chosen to comply.

### Quality System Procedures

The procedures in this quality system are the equivalent of work instructions in the ISO-9000 parlance. That is, they are step-by-step methods by which a process activity is accomplished. This level of detail in the procedure is valuable for gaining organizational consensus on how the activity is to be accomplished, and for consistency in the way multiple employees accomplish the activity. Note that this level of detail in a procedure may not be applicable to all projects that a discipline may experience. That is why this model for a quality system allows (even encourages) multiple procedures within the quality system for the same activity.

### Example: Tracing Unit-Level Testing Through the Quality System

As an example, let’s consider the activity of unit-level testing of software modules. As shown in Figure 4.3, a discussion (recommendation) of unit-level testing originated in FDA guidance document, in this case the GPSV. The company policy, consequently, requires the unit-level testing be performed at the beginning of the software testing as suggested by the GPSV. The software testing process of the company quality system further delineates the requirements for unit-level testing. As part of that delineation, the process requires that unit-level testing include test coverage metrics. Note that the process does not explain how those metrics are to be collected, or even calculated; it merely requires that metrics be collected and reported. The reason for this is that the methodology for collecting the metrics data could differ from project to project. Some projects may have software unit test tools that automate some of the coverage metrics. For other projects the only available option may be a manual means of collecting the same metrics.



**Figure 4.3** Tracing a requirement from regulatory guidance to quality system procedure.

As the quality system evolves, those in charge of the software testing discipline within the company may find it useful to compare metrics from project to project. If, and when, that is the case, some standardization of the metrics and means of calculation will be necessarily required at the process level. The means of collecting the data still would be defined at the procedure level where it can be changed to fit each individual project.

In the example of Figure 4.3 the procedure not only provides detail that a specific feature of the tool should be used to collect the metrics, but also sets the expectation threshold for what coverage is acceptable for the project.

In other projects that the software test group encounters, this procedure may not be appropriate. Other projects may not use the same computer language, or they may use another tool, or no tool at all. The thresholds of acceptability for path coverage may differ for other products on other projects. This is why the quality system suggested here allows for multiple procedures to accomplish the same activity for a process. It is during the planning for the project that the discipline leader considers whether procedures already exist that are adequate for the upcoming project. If they are, the plan simply refers to the preexisting procedure and the procedure that will be followed for the project. If no procedure exists, or procedures do exist but are not appropriate for the upcoming project, then the discipline leader has identified a

body of work that needs to be completed as part of the project. That is, a procedure needs to be developed for unit-level testing for the upcoming project.

The arrows in Figure 4.3 indicate some traceability from the regulatory guidance all the way through to the detailed procedure. Traceability is a useful tool for understanding the origin of requirements have become embedded in the quality system. Traceability is also useful tool for identifying gaps in the quality system; that is, for identifying regulatory requirements or recommendations that are not traceable to any requirement or activity in the quality system. There is plenty more to say about traceability in a quality system, but let's leave it at that for the purposes of this text.

In summary, the model above achieves the four objectives for a quality system as set out by the QSR's: organizational structure, defined responsibilities, documented procedures, and a review mechanism for approval and improvement of the system.

## Thinking Analytically About Responsibility

In the above discussion, it was briefly mentioned that a discipline manager was responsible for the process that defines his or her discipline. That is true, but is only a very superficial treatment of the subject of responsibility.

First, let's separate the responsibilities for the creation and maintenance of the quality system itself from the responsibilities for the design outputs (sometimes called artifacts) created by the activities defined by the quality system. The responsible parties for these two types of deliverables are likely to be very different. In the design of the quality system presented earlier in this chapter, a certain degree of flexibility was designed in to allow the designers, developers, and validation team to create project plans and procedures specific to the project at hand. That level of flexibility is likely to result in differing roles and responsibilities for each project. On the other hand, the roles and responsibilities for the quality system should be less flexible since quality system requirements are anchored in regulatory and industry standard requirements.

### Untangling Responsibilities, Approvals, and Signatures

Many companies bog down when it comes time to sign off (that is, approve) documents, whether they are quality system documents or simply design outputs. Much of this confusion arises from a lack of clarity about what the signatures mean. The confusion with signatures seems to be based in a lack of understanding about the individual's responsibility, and in the approval process in particular.

A tool that is frequently used to clarify roles and responsibilities is the RASCI chart. An example of a portion of one of these charts is shown in Figure 4.4. The rows of a RASCI chart present individual activities or deliverables. The columns of the RASCI chart represent the various participants in the project represented by the chart. The roles played by those individuals are indicated in the cells intersecting the rows and columns. The name RASCI comes from the abbreviations for the possible roles that individuals play. Those roles are explained in Table 4.1.

Discipline/Process/Procedure	President	Dir. Project Management	Project Managers	Dir. Internal QA/Regulatory	QA Engineers	Dir. Device V&V Testing	V&V Test Engineers	Dir. Systems Engineers	Systems Engineers	Dir. Software Development	Software Developers
Company Quality Policy	A	C		R	S	C		C		C	
<b>Project Management Discipline</b>											
Project Management Process		R	SI	A		I		I		I	
Job Estimation Procedure	A	R	SI								
Project Tracking Procedure	A	R	SI			I		I		I	
<b>Software Development Discipline</b>											
Software Development Process		I	I	A	I	A		I		R	SI
Software Requirements Procedure		I	I	CI	I	A	SI	CI	I	A	RSI
Software Design Doc. Procedure		I	I	CI	I	A	SI	CI	I	A	RSI
<b>Systems Engineering Discipline</b>											
Systems Engineering Process				A		I		R	SI	I	
Reviews & Approvals Procedure		CI	I	A	SI	CI	I	R	SI	CI	I
Risk Management Procedure		I	I	A	I	SI	SI	A	RSI	SI	SI
Defect Management Procedure		I	I	A	I	A	SI	A	RSI	A	SI
Configuration Management Procedure		I	I	I	I	A	SI	A	RSI	A	SI
Requirements Trace Analysis Procedure		I	I	I	I	SI	SI	A	RSI	SI	SI
<b>Software V&amp;V Testing Discipline</b>											
Software V&V Testing Process				A		R	SI	I		I	
System-Level Test Procedure		I	I	i	I	A	RSI	I	I	I	I
Integration-Level Test Procedure		I	I	i	I	A	RSI	CI	CI	CI	CI
Unit-Level Test Procedure		I	I	i	I	A	RSI	CI	CI	SCI	SCI
Test Trace Analysis Procedure		I	I	i	I	A	RSI	CI	CI	I	I

**Figure 4.4** Example of a portion of a RASCI chart.**Table 4.1** Definition of Roles Described by RASCI Charts

Role	Description
R	Responsible: The individual is responsible for getting the activity completed (the doer or coordinator of doers). The activity may represent the work of many people, but the responsible person is the one who will ultimately be held accountable if the activity is not complete.
A	Approver: An approver is one who reviews the results of the activity as part of the checks and balances of the quality system before it is put in practice.
S	Supportive: The supportive individual provides resource to those responsible and plays a supporting role in completing the activity.
C	Consulted: Those consulted provide necessary information toward completion of the activity, but are not responsible for completion, nor are they an approver of the result.
I	Informed: Informed individuals receive information about the completion of the activity, or information about the progress of that activity. They are not approvers, nor is their input a part of the completion of the activity.

Note that each activity should have one and only one responsible party. Activities with no Rs are gaps in the assignment of responsibility. Activities with multiple Rs have a responsibility ambiguity that may well have the same effect and the same end result as having nobody with assigned responsibility.

The RASCI or RACI (some don't use the supportive role) methods are well documented on the Web. A template RASCI chart is provided on the DVD that accompanies this text.

Figure 4.4 shows an example of a RASCI chart for the creation and maintenance of policies, processes, and procedures for part of a company's quality system. Note that this is not the RASCI for how the policies, processes, and procedures are actually executed. That RASCI belongs in a project plan. Clearly, this is incomplete and was fabricated as an example. Do not copy this as a complete RASCI that defines your entire quality system. Obviously, there are many ways that roles can be assigned. There are no rights and wrongs. It is a great start to simply be explicit about the roles and responsibilities for the activities. Use this as a starting point for your quality system or as a starting point for evaluating the roles in your current quality system. Analyze it, debate it, change it, use it. You will find out soon enough what works for your organization, and what doesn't. If it doesn't work, fix it next time.

The example is careful to assign responsibility to a single individual as discussed above. Assignments are made by position, not an individual's name. Why? Because individuals' roles change as they change positions or change companies over time. You do not want an individual's role cemented (inaccurately) in an obsolete RASCI chart, and you certainly do not need another document to maintain! Think about how an audit or regulatory inspection would handle role assignments to employees no longer with the company!

A concerted effort was made in the example to keep the approvals (As) for each activity to a minimum. Fewer approvals mean less delay in making necessary changes as the system evolves. The approval load was distributed as evenly as possible to make it more likely to get approved in a timely way. The company president only needs to approve three documents in this model. The director of QA and regulatory compliance carries the largest burden of approval with approval assignments for all quality processes, and those procedures that are critical to safety. That is because he/she bears responsibility for keeping the quality system in compliance with regulatory requirements.

Responsibility for creation and maintenance of documents is also distributed. The director of each discipline is responsible for his/her discipline's quality process. Note that the engineers reporting to a director have a support role to play in creating the process. That is, they will likely be assigned some responsibility for writing parts of the process, but the director bears the ultimate responsibility. Those engineers in the discipline clearly need to be informed about the process.

The systems engineering discipline in the company represented in this RASCI chart is responsible for many, if not all, of the cross functional activities as are itemized here. Consequently, more approvers are needed since the activities in systems engineering need heavy support from other disciplines in the organization. It seems rational that the directors of those disciplines upon which systems engineering relies

should have some approval of how their resources are being used in systems engineering procedures.

As the activities represented in this figure move down to the procedure level, responsibility for the creation and maintenance of those procedures is assigned to the engineers reporting to the director. Approval of these procedures is the responsibility of the director who is ultimately responsible for ensuring that each procedure meets the requirements of his or her discipline's process.

### **What Happened to the Author?**

Before getting away from the roles and responsibilities that are documented in a RASCI chart, two concepts deserve brief attention before moving on.

The first of these concepts is authorship. It is traditional for a company to require the signature (i.e., approval) of the author of the document. This requirement makes sense as a means of tracking the origin for the concepts contained in a given document. However, in today's work environment of group collaboration to create documents, it becomes increasingly unclear who the author of a document is. In fact, it creates uncomfortable situations to require the signature of an author whose document may have been altered significantly in the review process in ways that are unacceptable to the author. Consequently, the days of the single-authored document may be behind us. Thus requiring the signature/approval of an author adds little value to the document itself.

### **The Meaning of Approval: What That Signature Means**

The other concept to be discussed briefly is that of approval. Not all approvals on a document's coversheet need to indicate the same "meaning" of the approval. For instance, consider a software requirements specification. Suppose this specification requires the approval of the software engineering team lead, clinical specialist, marketing representative, verification test team lead, quality team lead, and project management. One can imagine that at some point in time these approvals were required for validating different perspectives on the product to be created. Software engineering approves the document as being correct and implementable. The clinical specialist and marketing representative approved the document as being clinically accurate and salable, respectively. The verification team approves the document as being verifiable. The quality team lead approves the document as being compliant with the quality system. The project manager approves the document as being implementable within the schedule and budget with which he or she has been given to work.

Although it seems obvious that approvers should approve within the bounds of their expertise, it is quite common for every approver of a document to feel that his or her signature implies that he or she has reviewed the entire document for all the above attributes. Reviewing for problems outside one's areas of expertise is difficult, time consuming, not productive—and easy to put off. In addition to being the cause of delays in approvals, it leads to reviews in which approving disciplines question the technical correctness of the responsible discipline's document. Imagine the mar-

keting or clinical team members questioning the design of a piece of software or a software engineer questioning the testing methods of the test team.

A simple suggestion for a review and approval procedure is to require that the signature page of any document include the meaning of the signature of each approver of the document. For example, for a quality system document such as a software development process or procedure:

*Director, Software Development:* Signature means that as the subject matter expert, the signer has read the entire document, agrees that it is complete and correct, understands its traceability from parent documents and to child documents, and agrees to execute the activity or activities as described in the document, or to provide adequate documented rationale for why it cannot be done so.

*Director, Quality Assurance:* Signature means that the quality reviewer has read the entire document, checked the traceability from parent documents for complete and correct traceability, and agrees that the document complies with the requirements of the parent document.

*Director, Marketing:* Signature means that as the marketing expert, the signer has read the entire document, agrees that it describes a marketable product, meets the needs of the target market, is compatible with other company devices, and is not likely to interfere with competitive device features that may be patented.

Maybe the wording is a little rough, but the point is to tell the approver (signer) of a document what you expect him or her to do for his or her approval and exactly what that signature means.

Still not convinced that signatures are that important? Examples of medical device manufacturer executives doing prison time may help convince you, but that is not a good learning example. Instead, let us look at an example of approvals gone awry that lead to the breakdown of a quality system.

## So, What Could Go Wrong with a Design Control Quality System?<sup>1</sup>

The following example is completely fictitious, and is not meant to represent any single company. Well, maybe it is not fictitious, but it is a fictitious blend of real situations we have encountered at our clients (and internally) over the years. Regardless, a healthy percentage of readers will be uneasy as they recognize elements of this story in what occurs in their own workplace.

AllBetter Devices, a fictitious medical device manufacturer, recently put in place corporate policies and procedures to comply with the FDA's design control regulations (21 CFR 820.30). The procedures outline the activities and documentation required to meet the regulation, and they detail the templates for the cover sheets for the documentation. Each document that is created must be "approved," and signatures are affixed to the cover sheet for each document to evidence the approval.

As the interested parties at AllBetter developed their design control polices, processes, and procedures, the topic of signatures and approvals came up for each

1. This example was first published in "Your Signature, Please: Approving Design Outputs," as published in Orthopedic Design & Technology, January/February, 2008.

document. To play it safe, AllBetter management decided that, with a few exceptions, all the company's stakeholders in the development of the device should be approvers on each design output. Certainly, they thought, this would be just what the regulatory agencies would want to see to convince them that the design and development process was well controlled, and that all interested parties in the company were in agreement with the design as it progressed through its life cycle.

AllBetter started its first product design and development project after gaining corporate acceptance of the new policies and procedures. The team was enthusiastic to use the new procedures and to finally "do things right." The systematic process would produce documentation at each phase of the development life cycle that would be reviewed, approved, and "signed off" before moving into the next phase. No longer would AllBetter allow itself to develop products in an ad hoc, random walk manner. They would never have to retro-document a project just before release to make it look like the company had followed a controlled design process. Nobody liked retro-documenting. It was boring, it seemed like a waste of time, and it didn't feel like it was an honest thing to do. This process should be so much better.

It didn't take long for the new design control process to begin to unravel. Engineering didn't want to approve and sign the Product Requirements Definition (PRD) until it had time to think about the requirements in more detail to be sure the device would be easy to implement. The team held off the engineering signature until it could think through some of the more detailed electronics and software requirements. After all, the engineering professionals didn't want to be on record that they had "approved" the product requirements only to find out later that they couldn't implement them. That certainly wouldn't look good for engineering.

The head of quality assurance for the project didn't want to sign the PRD until everyone else had signed. After all, knowing that the engineering team was still tinkering with the requirements and that it already was well into the next phase of the development life cycle put the team out of compliance with its own procedures. Withholding Quality's approval and signature was a way of telegraphing the quality team's displeasure with the way the project was progressing.

Marketing didn't want to sign the PRD yet. Two of the features that truly would sell the device were related to how fast the AllBetter-2 could actually make a patient, well...all better. Marketing was concerned that Engineering would change the PRD to make it easy to implement, but it would work no faster than the AllBetter-1. The company would never sell as many as marketing had forecast.

Almost everyone had a reason not to sign the PRD except the author, who felt the PRD was a real work of art and should be approved in record time. However, after Engineering, Quality, Marketing, and Clinical rewrote sections of the PRD, even the author didn't want to sign the PRD. It no longer was his document. It was not his writing, he didn't agree with the changes, and he didn't like their writing style. Signing as the author might mean that he would take the criticism in the future for poor thinking and poor writing in this document.

Six weeks later, the PRD was approved with all signatures affixed, but only after corporate management threatened to withhold annual bonuses unless the PRD was signed and the project moved to the next phase of the life cycle. Unfortunately, by the time the PRD was approved, the engineering team had already written about 80% of the detailed software and hardware requirements (or specifications). Several

of the engineers who were skeptical of the new procedures fell back to their old habits and started writing software before the software requirements had been approved—and they completely skipped the software design description required by their development process. Their rationale was that the PRD experience had confirmed their skepticism, so they would develop products as only they knew how. Managers looked at how long the approval process took for the PRD and extrapolated to estimate how long it would take for all documents required in the development process. They realized that the project would be two years late if they followed the approval process to the letter. They condoned the noncompliant behavior in the interest of getting a product to market.

A number of product requirements changes were made between the first release of the PRD for review and the version that ultimately was approved. Unfortunately, very few of the changes actually were implemented because engineering was “way ahead of those guys” and couldn’t “wait for them to make up their minds.”

Subsequent verification testing caught the fact that a number of requirements were missing from the product. Unfortunately, reworking the design to include these requirements was very expensive because the software was so complex. Changes late in development carried a risk that they might break something else that they didn’t intend to change. In fact, it was so expensive, the team decided not to include a number of product requirements just because of cost and schedule concerns.

Some changes were made, which, unfortunately, resulted in some defects that made their way into the field and led to a costly recall and regulatory attention.

## What Happened?

Weren’t these design controls supposed to prevent this kind of outcome?

Actually several things happened in this story: a process out of control, in-fighting and positioning among the development, quality and test teams, delays and asynchrony caused by delayed approvals. In this story, poor technical skills were not the problem. What triggered the sequence of events that lead to the unraveling of the design control process was the time it took to get the signatures on the very first document produced in the development process. Development teams are expected to work to meet scheduled milestones and releases. In many design control processes, they also are expected to wait for approval of the outputs of a given life cycle phase before proceeding to the next phase. It’s impossible to meet schedules and wait for approvals that cumulatively can take 50% or more of the development schedule to collect. It is common for development teams to discuss making corrections to a document, or even a company process or procedure—only to reject the idea because they don’t want to take on the approval process. This system becomes so restrictive that it defies improvement.

## Designing Streamlined RR&A Requirements for the Quality System

AllBetter Devices had several problems in its quality system related to approvals and signatures:

*Too many approvers and signatures.* The decision to “play it safe” and include all device stakeholders as approvers of all documents actually hurt the company. Many of the documents required approvals from disciplines that would not have much to contribute in the review of technical document. For example, what would a clinical specialist or marketing manager have to offer in reviewing a detailed software design specification? Requiring an approval in this case does not do much to build one’s confidence in the design output. Asking someone to do so probably would just result in procrastination.

*Unclear instructions and goals for approvers.* AllBetter’s procedures treated all approvers as equals. Either management expected a similar level of review from all approvers or thought the different approaches to the approval would be obvious to the different disciplines. Regardless, vague instructions for the approvers and unclear goals for their review and approval activities also encouraged procrastination if the approvers overestimated what was expected. If they underestimated what was expected, they would produce a poor outcome.

*Allowing delayed signatures.* Waiting for Engineering to determine what it could conveniently implement before setting the requirement was poor practice. Instead, the requirement should specify what is needed to meet the user (or other) need, not what is convenient to design. Certainly, holding up approval on a document for this activity should be forbidden. Too often, companies fear revising documents simply because the approval/signature process is so painful. AllBetter’s delayed signatures backed up the entire quality system, causing it to collapse.

*Approving and signing under duress.* AllBetter’s approvers finally signed off on the PRD when their bonuses were threatened. There is a good likelihood that some (or many) of them never actually reviewed the document before they signed it to protect their bonuses—even though they held up the process for their review. They may or may not have been aware of the potential personal liabilities associated with this activity. Regardless, they probably didn’t feel threatened, since each approver essentially was an equal. There is little value added in burdening people with approving documents that they won’t be able to critically review. Near the end of the life cycle, as device validation is underway, the group that helped define and approve the product requirements also should be involved in reviewing and approving the validation tests and results that confirm that the device meets the needs of the various stakeholders. In addition to the breadth of approver coverage, the corporate position of the approver also should be considered. Does a vice president need to approve every document in the design history file? A well-thought-out approval plan will place the signature responsibility at the level in the corporate hierarchy that the approver can critically review and approve the design output. At a minimum, executive management should be required to approve documents such as validation reports and risk management summaries so that they have a clear understanding of any residual risks in a device going to market. An approval plan also should include a definition of responsibilities and authorities that go along with the approval power. Some plans include an escalation process for when consensus cannot be reached among the delegated approvers.

*Failure to provide training for the approvers.* Approvers should be trained to understand the potential impact on the device design, and ultimately, on the users of the device if they don't take their approval responsibilities seriously. Approvers also need to be made aware of the potentially destructive effects of regulatory action on the company and on them as individuals if they fraudulently sign a document, or if they allow design errors to slip through because they don't take their approval responsibilities seriously. The intent of this training is to inform approvers of how the quality system depends on their expertise in the approval process and to help them understand how to fulfill that expectation. In AllBetter's case, each participant depended on "the other guy" to do the review.

*Allowing "signature sniping.* Let's face it—demanding to be the last signer on a document is a power play. Approval of design outputs is a serious business and should be above the gamesmanship of arguing over whose opinion or time is more valuable.

## Fixing the Problem: Designing a Value-Added Approval/Signature Process

There may be some art and science involved in creating a good approval process, but mostly it is just common sense. Careful deliberation over the following points will lead to a streamlined approval process that will actually add value to the development process and will not become the bottleneck that destroys the quality system.

Design an approval procedure. Few documents need to be reviewed by everyone, and few approvers need to approve every document. Don't require an approval from a stakeholder unless that stakeholder will have valuable domain knowledge to contribute to the design output. In general, early phase documents related to the definition of the product will benefit from a larger list of approvers. Detailed technical documents will have a much smaller list of stakeholders—those who will contribute valuable feedback and will understand the document. Don't add approvers to "play it safe." Too many approvers can destroy your process. The design of an approval plan can be thought of as an assignment of responsibility for the design of the product. Clinicians and marketing departments with knowledge of the clinical, market, and user needs should approve the product-level requirements. Regulatory Compliance, Legal, Service, and Engineering all will have their opinions about the product requirements, too. In the middle phases of the development life cycle, few groups other than engineering will have an opinion about the potentially devastating effects of an inadequate review process on end users, the company, coworkers, and the approver him- or herself. By giving the approver a clear sense of responsibility for the approval process, he or she may be more likely to treat the responsibility seriously and less likely to sign under duress.

Set clear expectations for what the quality system expects from each approver. It is quite possible that no two approvers should look at a design output for the same thing. Make it explicitly clear to each approver what his or her signature is assumed to mean. Consider adding a short statement next to each approver's signature that clarifies for what aspect of the approval the signer is taking responsibility. For

example, next to the marketing department's signature the signature page may state: "Marketing approval implies that this document has been reviewed for meeting the needs of the market, for competitiveness and salability of the finished product." Note that marketing is not expected to review the document to determine if the design is error-free, or if it meets all clinical needs, or if it interferes with competitors' patented features. Presumably that level of review is being done by other approvers. The meaning behind the signatures will depend on who is on the approval team.

By making approvals part of a well-defined procedure, it will help eliminate approval delays. An approval procedure should allow adequate time for the approver to review the design output on his or her own. A review should be held with all approvers to compare their individual review notes. Action items should be identified that will resolve all issues raised by the approvers. The individual/group review process may need to iterate but should end in a final approval meeting where all approvers sign the document at the same time. This eliminates "signature sniping" and changes a nearly serial approval process into a parallel process that takes less calendar time.

## Regulatory Basis for Treating Approvals and Signatures Seriously

Let's assume that a company has been organized to support the effective implementation of its quality system as it relates to design, development, production, and any other aspect of the quality system. Let's narrow our focus to the design controls section of the QSRs (820.30) that relates most closely to software validation. Figure 4.5 summarizes the specific documentation, review, approval, signature, and date-stamping requirements of the design control regulations.

Additionally, the design controls require that "each manufacturer shall establish and maintain procedures to ensure ..." the appropriate and correct execution of each activity specified in the design controls. Although review, approval, signatures, and date stamps are not specifically required for each of these procedures, one can assume that they are intended from the document controls section (820.40) of the QSRs.

Design Control Activity	Documented	Reviewed	Approved	Signature	Date
Design and development planning	Implied	Yes	Yes	-	-
Design input	Yes	Yes	Yes	Yes	Yes
Design output	Yes	Yes	Yes	Yes	Yes
Design review	Yes	Implicit	-	-	Yes
Design verification	Yes	-	-	-	Yes
Design validation	Yes	-	-	-	Yes
Design transfer	-	-	-	-	-
Design changes	Yes	Yes	Yes	-	-
Design history file (DHF)	Implied	-	-	-	-

**Figure 4.5** Regulatory requirements for documentation, review, approval, and signature.

**820.40 Document controls.**

Each manufacturer shall establish and maintain procedures to control all documents that are required by this part. The procedures shall provide for the following:

(a) Document approval and distribution. Each manufacturer shall designate an individual(s) to review for adequacy and approve prior to issuance all documents established to meet the requirements of this part. The approval, including the date and signature of the individual(s) approving the document, shall be documented. Documents established to meet the requirements of this part shall be available at all locations for which they are designated, used, or otherwise necessary, and all obsolete documents shall be promptly removed from all points of use or otherwise prevented from unintended use ....

The “part” referred to in this regulation is Part 820, the quality system regulations. Consequently, even though reviews, approvals, signatures, and dates are not specifically required for any individual document described in the design controls regulation, one can infer from 820.40 that the Agency does expect systematic and documented reviews and approvals for any required documentation that is part of the quality system.

A signature in the context of a design control process simply is hard evidence that the person to whom the signature belongs has “approved” the document to which the signature is affixed. The signature is secure in the sense that it is not easily copied and usually can be detected if someone tries to copy it. (Electronic signatures pose special problems in this context and are regulated by Part 11 regulations.) Let there be no confusion. The signatures required by these regulations are legally binding signatures. The dates are used by regulatory inspectors in their assessment of compliance to a device manufacturer’s quality system. Any backdating of retro-documentation is clearly in violation of the regulations, and frequently can be detected by inspectors. The quality system inspection process of the FDA will not be covered in any detail in this text, but the FDA’s “Guide to Inspections of Quality Systems” [1] that is included in the DVD that accompanies this text makes it clear that dates and signatures are treated very seriously during quality system inspections.

## Reference

- [1] *Guide to Inspections of Quality Systems* (also known as QSIT—Quality System Inspection Techniques), U.S. Food and Drug Administration, August 1999.



# The Software (Development) Life Cycle

The software life cycle, or software development life cycle (SDLC), has been referred to a number of times in earlier chapters of this text, and is often referenced in the FDA guidance documents. In this chapter, the focus will be on the software development life cycle itself; what it is, why it is important and adds value, and how to choose a life cycle that is appropriate for a given software item as it relates to validation.

## What Is a Software Life Cycle?

A software life cycle, in general, is simply the recognition that the existence of a software item within an organization goes through distinct phases. There are common characteristics of all life cycles, or at least that is true for the purposes of controlling the life cycle through verification activities discussed in Chapter 6. The common characteristics are:

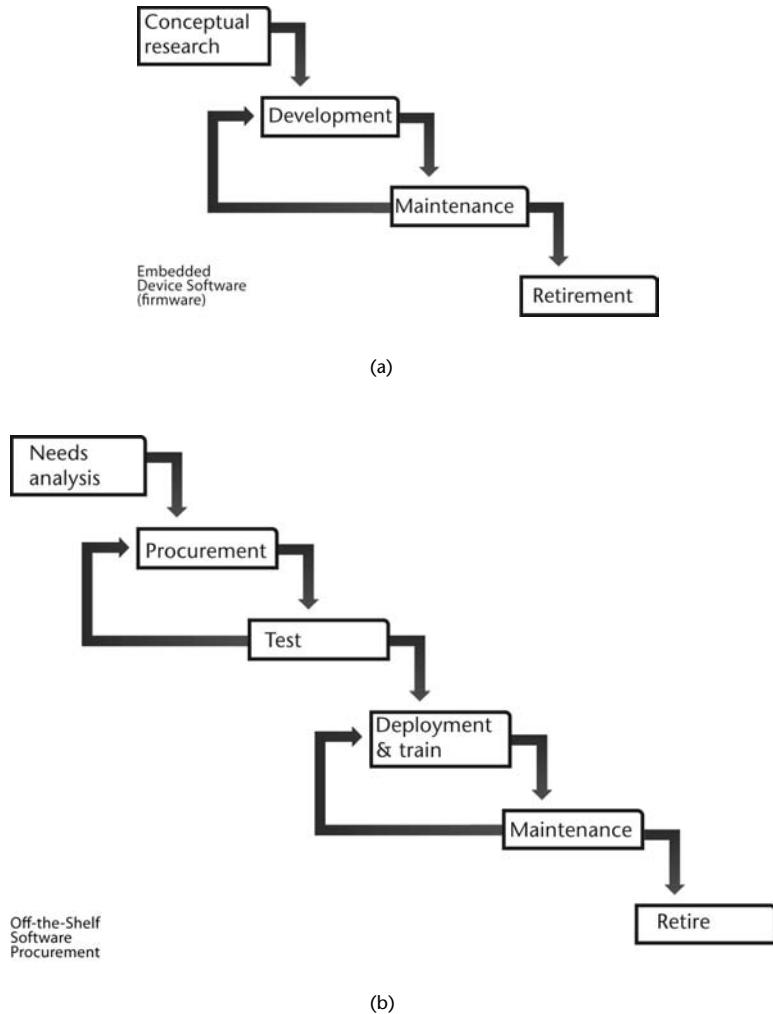
- Identifiable phases;
- Inputs to each phase;
- Activities to be performed in each phase;
- Outputs of each phase.

Without these characteristics, the control of the design (which is after all the intent of the design control regulations) is without easily identifiable milestones, metrics, or control points. How many phases; what they are called; how they transition; what takes place inside each phase—these are all up to the device manufacturer to determine.

Life cycle models come in all shapes and sizes. Some variety is necessary to accommodate the diversity of sizes, complexities, and usage models for software. This will become especially evident in the Part III of this text, which deals with the validation of quality and production system software.

Figure 5.1 shows a comparison of two high-level life cycle models for two very different software items. The top panel of the figure shows a fairly standard high-level model of the software life cycle for software that is developed by the device manufacturer to be part of a medical device. The bottom panel shows a high-level life cycle model for software item to be purchased off the shelf. The software might be used as part of the production process, or as a tool for development.

It is clear from comparing these two panels that the activities and controls over the life cycle would be very different. In the case of the medical device software, emphasis is on the development and maintenance of the software. In the case of the off-the-shelf software, the emphasis is on acquiring and testing software for suit-

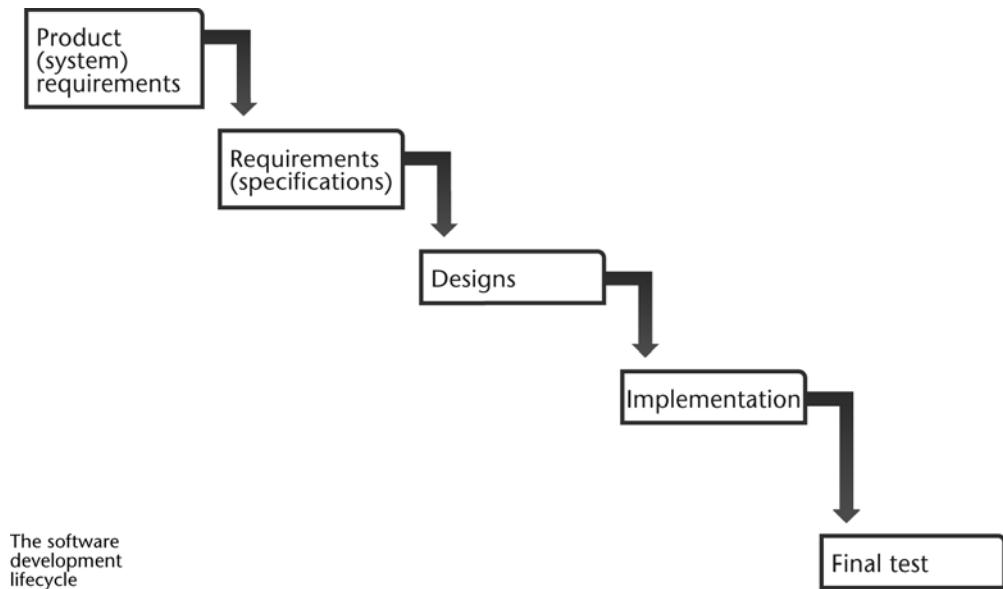


**Figure 5.1** (a, b) High-level life cycles of two very different types of software.

ability, followed by training and deployment of the software. Both are models of life cycles for software. The SDLC that is often referred to really only applies to the development phase for software that is custom developed.

Unfortunately, many companies try to force the adoption of a single software life cycle for the entire company for all software. The comparison of the two models of Figure 5.1 should make it clear that any effort to make either model apply to all software could lead to frustration as the software team struggles to fit their software life cycle into a model that does not quite fit.

Figure 5.2 shows one example of how the development phase of the overall life cycle can also be broken down into subphases. These are collectively referred to as the Software Development Life Cycle or SDLC. It would be difficult to argue the existence and sequence of phases at the highest level as shown in Figure 5.1. However, the definitions of the subphases under the development phase and the sequencing of those subphases are the subject of much debate within the development community.



**Figure 5.2** One view of the development life cycle hierarchy.

A number of different SDLC models will be discussed shortly, but for now let us leave it at this: there are few absolutely right and absolutely wrong choices for SDLC models. Perhaps the only wrong choice would be to try to standardize on a single SDLC model for all situations related to software development and validation. For example, the SDLC model shown in Figure 5.2 is often referred to as the waterfall life cycle model simply because it is usually graphically represented as it is in Figure 5.2, which somewhat resembles a cascading waterfall. This is perhaps the most common of the SDLC models, and seems to fit intuitively with the sequence of actions one could anticipate for designing and developing a new device from scratch. However, imagine what the sequence of activities might look like for a second release of the embedded device software. Oftentimes for later releases of device software, the design documents are not totally rewritten nor are the user needs and product requirements for the entire device reexamined. Instead, an abbreviated form of the waterfall SDLC might take place that focuses only on the changes to the requirements, documentation, software, and validation activities. There might also be additional activities included in such a project such as regression testing of the software to be sure that there are no unintended consequences of the changes that are being made.

The point is that, like so many other things related to software development and validation, our propensity to standardize seldom considers all of the situations to be encountered in the future. In fact, it may even be impossible to do so. So, by standardizing on a single SDLC model, a company will force those attempting to comply with that standard to fit their projects into that SDLC model whether they are appropriate or not.

## Software Validation and SDLCs: The Regulatory Basis

If you are life-cycle skeptic, you may be asking yourself whether any of this life-cycle business is rooted in regulation. Actually, the term life cycle or lifecycle does not appear in the QSRs at all. There is reference in the QSRs to “stages of the device’s design development,” so clearly there was regulatory thought about the design process having well-defined stages.

The General Principles of Software Validation (GPSV), which expands on the QSRs, does introduce the term “software life cycle” in Section 4.4:

Software validation takes place within the environment of an established software life cycle. The software life cycle contains software engineering tasks and documentation necessary to support the software validation effort. In addition, the software life cycle contains specific verification and validation tasks that are appropriate for the intended use of the software. This guidance does not recommend any particular life cycle models—only that they should be selected and used for a software development project.

So, while the FDA does not mandate or even recommend a specific SDLC model, they clearly are of the opinion that an SDLC model should be selected and used. Why regulators may have come to that conclusion will be discussed shortly.

The FDA does provide some guidance on what activities they expect to be covered by the SDLC model that is selected. General Principles of Software Validation provides some further insight into the agency’s expectations in Section 5.1:

### SOFTWARE LIFE CYCLE ACTIVITIES

This guidance does not recommend the use of any specific software life cycle model. Software developers should establish a software life cycle model that is appropriate for their product and organization. The software life cycle model that is selected should cover the software from its birth to its retirement. Activities in a typical software life cycle model include the following:

- Quality Planning
- System Requirements Definition
- Detailed Software Requirements Specification
- Software Design Specification
- Construction or Coding
- Testing
- Installation
- Operation and Support
- Maintenance
- Retirement

Verification, testing, and other tasks that support software validation occur during each of these activities. A life cycle model organizes these software development activities in various ways and provides a framework for monitoring and controlling the software development project.

## Why Are Software Development Life Cycle Models Important?

Why? Without a life cycle model, or defined stages of development, it is difficult to know where software development or validation project stands or even where the “control points” are. Without some predefined staged development and/or validation plan (i.e., SDLC model) an outsider (say, the FDA) might have difficulty believing that the design process was “controlled” ... and they would probably be right.

Why is a controlled design process important? Recall the “two prongs of building confidence in software” discussed in Chapter 3: keeping defects from ever manifesting themselves in the software, and finding and removing those defects that do. Design controls are necessary to reduce the probability that defects will be embedded in the medical device; that is, the first prong. Our intuitive logic tells us that a controlled design process results in better designs and implementations. Granted, sometimes projects can be overcontrolled and do not improve the outcome at all, but that is no reason to reject the design controls concept *in toto*.

So, life cycles are part of the design controls, and design controls are necessary to reduce the probability of defects being designed into device software. In addition to satisfying regulatory requirements, there are a number of benefits to the business that are realized by following a defined SDLC model. Some of these benefits include:

*Measure before you cut.* The old woodworking or carpentry axiom of “measure twice, cut once” applies to software. Planning, reviewing, and revising in the early stages of a software project can save time and money while improving the quality and transparency of the software development process. Avoiding costly “recuts” of the software benefits the business. Planning, reviews, and controls improve the quality. Transparency of the development process allows for more productive verification and validation processes.

*Control.* Perhaps the primary benefit is control of the software development and validation process. Control not only improves the quality of the resulting software, but can also improve the economics of the project (i.e., schedules and budgets). As the development life cycle transitions from one phase to the next, it provides an opportunity for review of the outputs of the phase just completed and the adequacy of those outputs as inputs to the next phase.

Why is that important and what are we looking for? From the quality perspective, the review should ensure that the user needs and product requirements have been correctly translated and carried forward in the activities of the phase just completed. As we will see in the next chapter, the GPSV and the design control guidance would term this a verification activity. From a development perspective, the outputs should be reviewed for completeness and correctness as well as their adequacy as inputs to the next phase. From a regulatory perspective the outputs should be reviewed for their compliance with regulatory requirements and guidances. From the business perspective, project management assessments should consider the historical experience of past phases, and update projections for future phases based on the results of the work completed. This gives the business control by managing the

project risk based on an assessment of how well controlled earlier phases were, and how much upcoming phases have expanded beyond the initial estimates.

*Metrics.* There is no such thing as a perfect metric when it comes to software, but *some* metrics are better than *no* metrics. Project metrics are probably of most value to the business. Collection of metrics on a phase-by-phase basis provides an indication of whether a project is under tight control and tracking well to estimates, or is a runaway project that is exceeding cost estimates and making too little progress. Metrics provide a useful view of the project at the end of the development life cycle by providing insight into how much the requirements changed throughout the life cycle, how many defects were discovered at each phase of the life cycle, the relative costs of each phase of the life cycle, and the actual to estimated ratios or costs and schedules. Why is that important? It is important to have some high level understanding of what is working well, and what is not, so that the development/validation processes can be improved over time.

*Organization.* For the engineers contributing to a software project, the mapping of activities that are required by the company's quality system and/or the regulatory agencies into life cycle phases provides an organization for the project and a check list of those activities and outputs to be accomplished at each phase of the life cycle. Effort expended on activities outside those of the current life cycle phase or subphase are indicators of a project that is running out of control. That could have consequences not only for the business, but also for the quality of the resulting device. Likewise, activities within the current phase that are logging no effort are indicators of a process problem or gap. Breaking a large project into smaller phases makes it easier to understand the current status of the project.

There are probably a number of other good ways to explain the value of life cycle modeling. Perhaps the easiest explanation I have encountered is simply this: breaking a development or validation project into phases makes it easier to explain to others how the work is to be done, or how the work was done (in the case of an audit or inspection). If one has trouble explaining a development plan, chances are there is a lack of understanding of that plan. Life cycle models are like block diagrams. They are tool to help us break down complex concepts to make them easier to understand.

## What Do Different Software Development Life Cycle Models Look Like?

For whatever reason, selection of an SDLC model seems to be an emotional experience for many companies. No life cycle model is perfect. They are simply frameworks for planning the activities for development and validation. It is nearly impossible to rigidly adhere to any simple life cycle model for software of any moderate size or complexity. The objective should be to select an SDLC model that best fits the project at hand, then manage the deviations from the life cycle to a minimum. It is simply a matter of balance and common sense.

### Waterfall and Modified Waterfall

One of the first documented life cycle models is the waterfall life cycle model. One of the advantages of this model is that it is simple and intuitive. As seen in Figure 5.3, the waterfall model is one of the linear SDLC models referred to above. The phases move forward in a one-way manner that put requirements before design, design before implementation, and implementation before test. This is an intuitive progression of phases; however, it does not reflect the iterative nature of software validation and development for even moderately complex software.

The modified waterfall SDLC model as shown in Figure 5.4 shows some of the feedback pathways that reflect the normal revision and correction necessary in software development as requirements change and as errors are discovered. Even this model is somewhat unrealistically simplistic. For example, it is not uncommon to discover defects in the final test phase of the development life cycle that require changes to requirements and specifications. The modified waterfall shown here only depicts life cycle pathways back to the preceding phase. While more accurately accounting for the iterative nature of software development and validation than the traditional waterfall model, the simplified feedback shown in the modified waterfall is also unrealistically simplistic.

### Sashimi Modified Waterfall Model

The catchy name for this SDLC model comes from the Japanese dish sashimi, which is often served as overlapping pieces of fish. This modification of the waterfall model allows for overlapping phases of the life cycle. In other words, while some of the requirements are still under development, other parts of the project may be in design, implementation, or even test depending on how much overlap is allowed or is allowed to evolve.

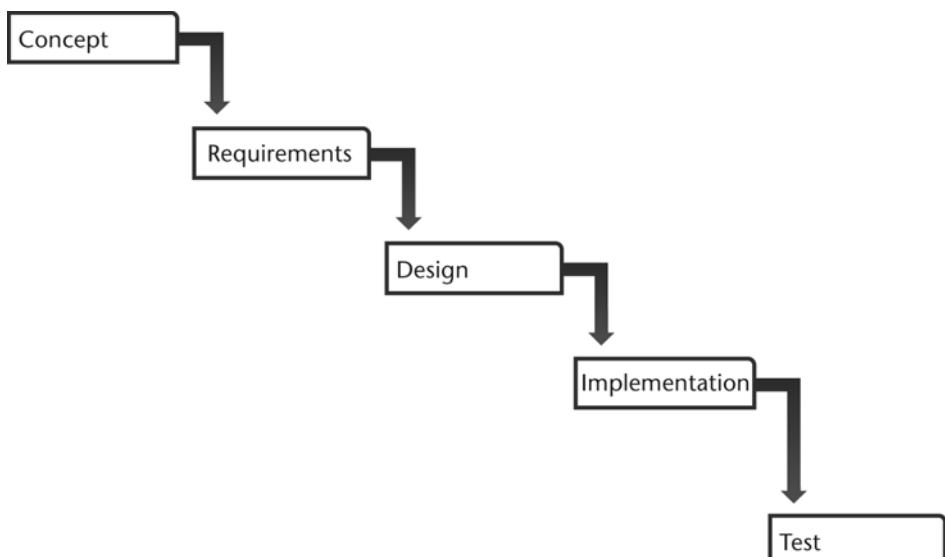
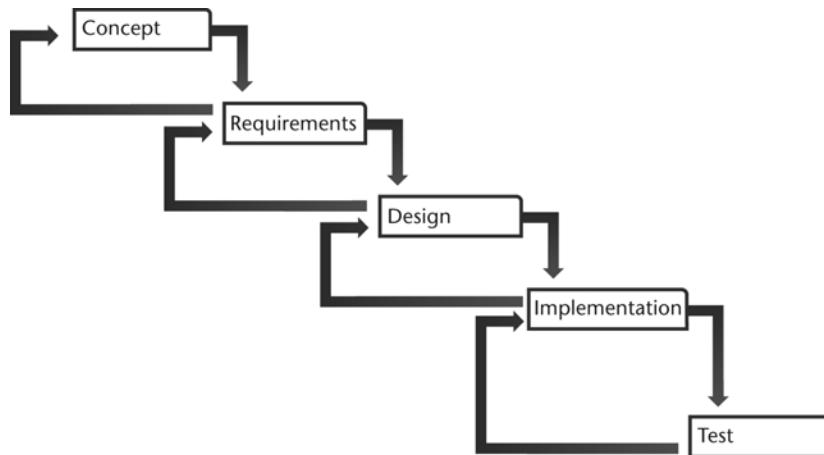


Figure 5.3 Waterfall SDLC model.



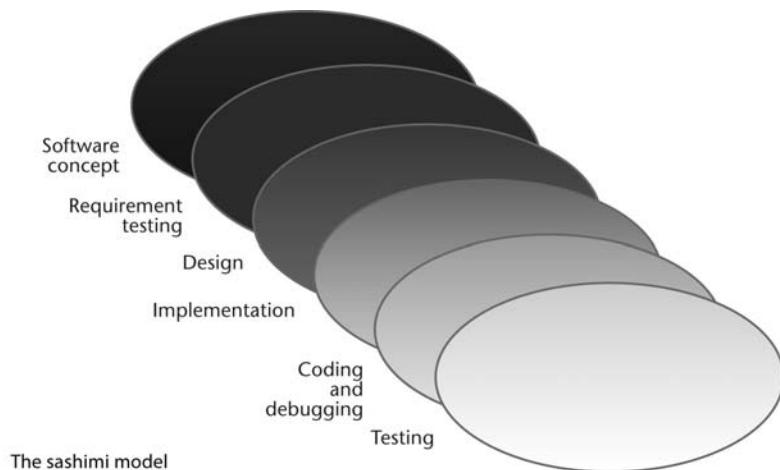
**Figure 5.4** Modified waterfall SDLC model.

This model is interesting because it represents a turning point in this discussion from life cycle models representing the way management and regulatory agencies would like for the SDLC to progress, to life cycle models that are more representative of the way most projects are actually done, or the way most software developers would like to do them. (Note that a process is not necessarily correct, or better than other options just because it is what was done in the past, or because it is the preferred process.) As we will see in the discussion of this model, the spiral model, and the agile model, the control points are less obvious, the metrics are more difficult to understand, and the use of the life cycle as a gauge for progress and forecasting is less likely.

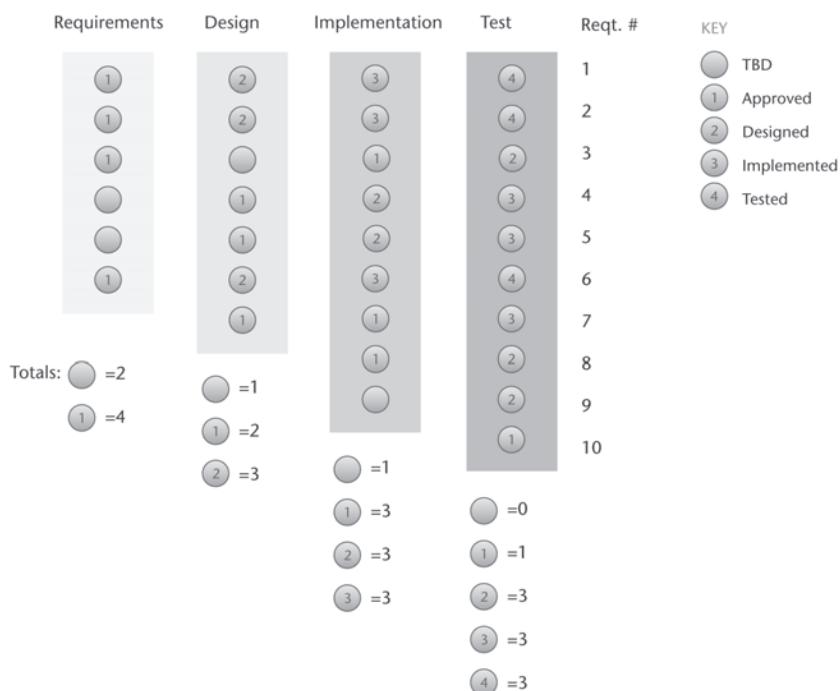
Figure 5.5 shows a graphic depiction of the sashimi modified waterfall SDLC model. The size of the phases and the amount of overlap will have a huge impact on the way the product is developed and validated. Imagine the chaos if every function of the software were simultaneously allowed to be in the requirements, design, implementation, and test phases. It would be nearly impossible to synchronize the design documents, tests, and software. This is precisely the chaos that a regimented traditional waterfall tried to correct by forcing formal phase boundaries.

Can the sashimi modified waterfall be managed in a way that allows for the realities of product development and validation while not contributing to the lack of control that the life cycle is supposed to address? Perhaps. Consider the example of Figure 5.6. In this figure, each of the columns represents the state of the software requirements in each of four phases of the sashimi modified SDLC. It is not clear whether these representations are taken from midphase or end-phase since the phase boundaries are unclear. Be that as it may, the numbers enclosed by circles represent the status of each requirement in the requirements document.

In the requirements phase, all but two of the requirements have been written and approved. The team recognized the need for two additional requirements but did not have enough information to detail the requirements. Consequently, they were left to be determined (TBD). In the next phase, theoretically the design phase, the approved requirements of the preceding phase were released for design. In the pro-



**Figure 5.5** Sashimi modified waterfall model.



**Figure 5.6** Requirements evolution under a sashimi model.

cess of designing requirement number 3, the team recognized an inconsistency in requirements and requirement number 3 was returned to a TBD status. The team also discovered the need for an additional requirement and it was added as requirement number 7.

The project entered its implementation phase. The three designed requirements were implemented in software. The new requirement number 7 was designed. The inconsistent requirement number 3 was rewritten and approved. Unfortunately, the team found a need for three new requirements (numbers 8, 9, and 10). Two of these new requirements they proposed, and one (number 10) needed additional information and was set to a TBD status.

The next phase was the test phase. Each of the 10 requirements advanced status by one step. However, in the middle of the test phase, we find ourselves with one requirement not yet designed, three requirements designed but not implemented, three requirements implemented but not tested, and only three of the 10 requirements actually tested.

Of course the test phase was not complete, and the team could continue to iterate through requirements refinement, design, implementation, and test but is it appropriate to think of the development life cycle in its test phase when only 60% of the requirements are even implemented? Perhaps the team transitioned through the phases too quickly and the requirements document represented in the “Test” column actually represents the third release of the requirements document in the design phase. In that case, is it appropriate that 60% of the requirements are already implemented and/or tested while the software is still being designed?

It seems possible that this sashimi modified model can be made to work if an “atomic level” requirement by requirement status is managed. The names of the phases with such indistinct boundaries are meaningless, and might as well be called A, B, C, or January, February, March (and maybe that is a good idea).

On the positive side, the life cycle model portrayed in the example of the preceding figure is at least under some level of design control. Requirements are written and approved before they are designed and implemented. To be sure, even in the most strictly enforced traditional waterfall models some level of activity like this has to take place. Few if any design/validation teams are good enough to write all requirements perfectly the first time, or to design them all perfectly before they are implemented. However, openly planning, and encouraging overlapping life cycle phases will likely lead to delayed decisions, more churning of requirements, and more rework necessitated by functionality whose requirements and designs are only finalized late in the life cycle. There are situations in which the sashimi model may be very appropriate for requirements and designs that, by necessity, cannot materialize simultaneously. There also situations in which the sashimi model is chosen simply because it most accurately models the current lack of process within an organization.

## Spiral Model

There are situations in which little is known about the requirements for a device or for device software. In these situations, it is advantageous to select a life cycle model that allows for early and rapid prototype development to get the device and/or software into the hands of users early so that user feedback can be used to refine the requirements for the final product. A spiral life cycle model is often selected for situations like this.

There are many different versions of spiral life cycle models. The example shown in Figure 5.7 is an adaptation of several spiral models proposed for medical device software development and validation purposes.

Each of the four quadrants of this figure represents major activity groupings: Planning, risk assessment and management, development, and verification and validation testing. The timeline for the life cycle starts at the origin of these four quadrants. We have chosen to begin the process with a preliminary hazard analysis (PHA) in the risk assessment quadrant. Each outer ring of the spiral represents a release of the documentation, the software or the device. This spiral is very simple, but one can readily appreciate that moderately complex projects might have many more rings in the spiral. By progressing clockwise around the spiral, each of the four activity groups is represented for each successive release of the software. Note that within the development quadrant, different rings of the spiral might actually use different subdevelopment life cycle models. For example, inner rings of the spiral in which the concept is still being tested with the intended user base, the development

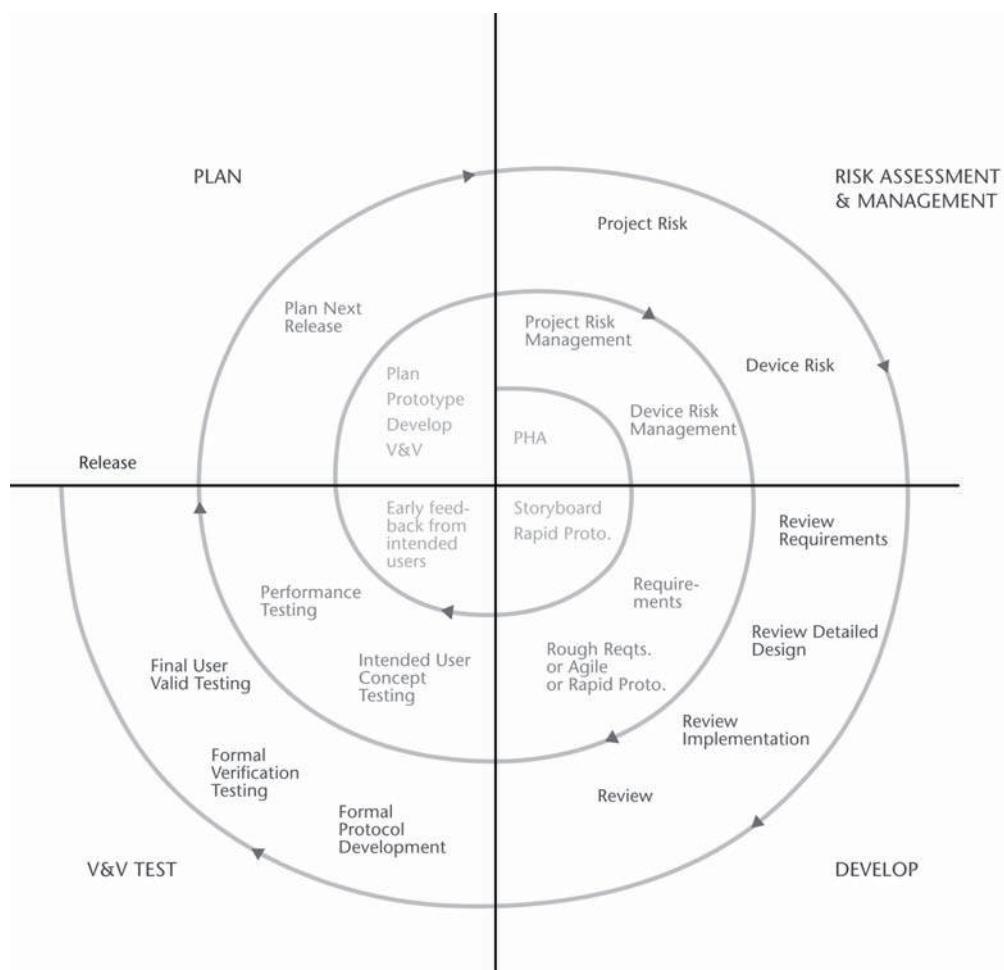


Figure 5.7 Spiral SDLC model.

might actually be done with storyboarding, rapid prototyping tools, agile methods, and so forth, while the final production release of the software might be developed under a traditional waterfall model. At that point in the SDLC, enough should be known about the requirements and design of the software that a traditional waterfall should be more easily achievable.

A spiral model is generally not a good choice for small projects, projects whose requirements are well known and understood, or for development projects whose testing of intermediate releases of the software would put the intended user base testing the software in harm's way.

### **Extreme Programming: Agile Development Models<sup>1</sup>**

The agile movement is still relatively young. Agile methods are a collection of associated methodologies covering the design, development, management, and testing of software. Among the most widely touted of the agile methods is extreme programming, or XP. It is perhaps the most visible and controversial of the agile methods. XP is often referred to as “lightweight” software development because it sidesteps the creation and maintenance of detailed software requirements or detailed design documentation. Programmers work in small groups that are further broken down into pairs. Each pair shares a computer. One person writes code while the other audits as a means of reviewing and cross checking each other’s work. The software evolves through definitions of requirements, called stories, which are short enough to be described on index cards. The process is intended to be rapid, with software examples quickly being written and presented to the customer for either acceptance or incremental change to the story requirements.

Traditional techniques try to discourage change in the late phases of the development life cycle due to increasing costs in maintenance of the software and its inter-related requirement, design, and test documents. The XP methodology, on the other hand, encourages constant change and theoretically responds to change faster because its lightweight nature carries no supporting documentation to update with changing source code.

“Scrum” is a project-management method whose name is borrowed from rugby. The basic concept of scrum is to organize around short-term and long-term goals. The long-term goals, called sprints, are expected to be accomplished in 30 days of development activity. Short, stand up meetings, or scrums, of 15 to 30 minutes are held daily to review the progress of the previous day and to set goals for the upcoming day.

Test driven development (TDD) has more recently evolved as an agile method. It sounds like an appealing methodology, given the software verification and validation regulations for medical device software. TDD focuses on developing tests for the software items before the software is written. The tests are automated so that they can be rerun in their entirety on a build-by-build basis. The TDD sequence is: develop tests for the software functionality, write the software, run the tests, fix the

1. For more thoughts and opinions on agile methods and medical device software, see Vogel, D. A., “Agile Methods: Most Are Not Ready for Prime Time in Medical Device Software Design and Development,” DesignFax Online, Vol. 2, No. 27, July 25, 2006.

software and refactor. Refactoring is an agile concept that requires redesigning or rewriting software to correct for poorly partitioned or redundant code modules.

The agile methods interest groups are to be applauded for their work in experimenting with new software development techniques. As mentioned earlier in this chapter, software is so broadly defined that different types of software may require different development methodologies and life cycle models, just as they benefit from different development languages, tools, operating systems, and so forth.

However, it may be too fanciful to expect that a regulatory agency charged with responsibility for inspection of design activity would agree to less (let alone no) documentation of software requirements and designs—at least under existing regulations and guidance. However, the methodologies are intriguing, and some methodology is better than none. The key to widespread use of agile methods in the medical device industry will be to reconcile the activities and outputs with regulatory expectations. At the time of this writing, a workgroup has been convened by the Associations for the Advancement of Medical Instrumentation (AAMI) for the purpose of writing a technical information report (TIR) for the industry on the topic of the use of agile methods for software in the medical device industry. Perhaps that group will find a way to make it all fit!

## How Do You Know What Life Cycle Model to Choose?

Requirements change, mistakes are made, microprocessors and tools go obsolete, and one simply cannot predict everything. Consequently, life cycle models that model the process as a straightforward linear progression are probably not realistic if one expects a fundamentalist level of adherence to the model.

On the other hand, there is potential danger in admitting from the outset that the life cycle cannot be followed verbatim. This creates an environment for team members to all interpret the life cycle as it pleases them. Furthermore, once development and validation teams get a sense that there is little value in an activity (like shoehorning a development/validation project into a life cycle that doesn't fit), they get very creative in their ways of avoiding it, proving it is wrong, or even sabotaging the activity.

There is a chance that selection of a software development life cycle can actually slow down the process of software development and validation, and not lead to any quality improvements at all if the life cycle does not fit the reality of the project, or the project team members are not believers in the life cycle model. The theoretical improvement in the process can lead to failure in much the same way as the implementation of the design control quality process failed in the example of the previous chapter.

With that dire outlook, let us consider what can be done to improve the odds.

1. Select a life cycle model that closely matches the realities of how the software needs to be developed and/or validated. For example, for cutting-edge technology for products that are unlike any other currently available, discovering the requirements may well be an iterative process, whereas when developing software for the third generation of the device whose

requirements are well understood, a more linear model may be quite acceptable and more efficient.

2. Understand that the life cycle model is a model for project planning, not a directive that must be followed at all costs whether it makes sense or not. Flexibility makes good sense, but some controls will be necessary to avoid changing life cycle models every week.
3. Be flexible. There isn't going to be a glass slipper life cycle for your project. Be prepared to make adjustments while sticking to the basic framework and timeline of your chosen life cycle.
4. Don't be afraid to choose different life cycle models for different software projects. Not all projects have the same legacies or challenges. Fit the life cycle model to the project at hand.

Table 5.1 summarizes some of the pros and cons of the different life cycle models discussed in this text. It may be useful in selecting a life cycle model for a future project. The models discussed above by no means represent all the possibilities.

## How Do Software Development Life Cycles Relate to the Quality System?

So far in this text, we have covered the quality system and the policies, processes, procedures, and plans that comprise the quality system. We are now nearing the end of our discussion about life cycles. This book is about validation. How does this all fit together?

We discussed in this chapter the benefits of recognizing the software life cycle, and more specifically the software development life cycle. By now it should be clear that the interest the FDA has in life cycle management is to build confidence in the software by using good engineering practices and controlling quality by a formal review process at each phase of the life cycle. (Medical device developers should share the same interests regardless of the FDA's interests.) We build confidence by preventing defects through good engineering process. That's important because it is easier, more efficient (and less expensive) to prevent defects than to find them through testing.

So, how does the life cycle fit with the company's quality system? There are four ways that are immediately obvious:

1. Organize each discipline's quality process (as defined last chapter) around a common life cycle model. In other words, software engineering, systems engineering, software quality assurance, and other company disciplines organize the activities of their processes in a common SDLC model (e.g., traditional waterfall) for all software projects. This has the benefit of everyone in the company using the same language to describe life cycles, and developing a level of comfort with the phase definitions. (This is what we chose to do at Intertech years ago.) The disadvantage of this alternative is that not all software projects neatly fit a single SDLC model, and there will always be some tailoring of the project plan to fit the life cycle model. This

**Table 5.1** Advantages and Disadvantages of Several SDLC's

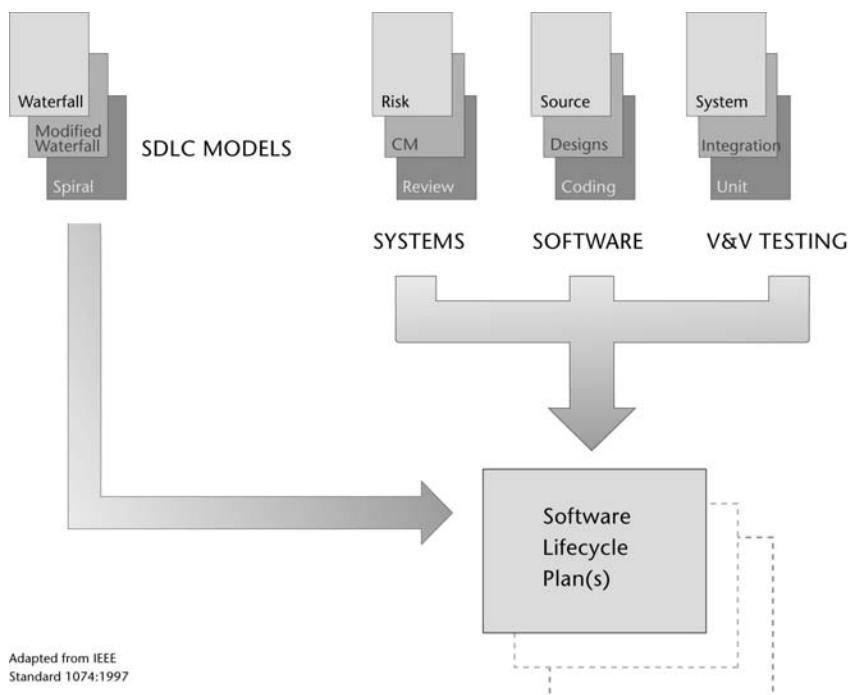
Pros	Cons
<b><i>Traditional Waterfall</i></b>	
Intuitive, easily understood. Established and well known. Enforces design before implementation. Control points clearly demarcated. Leaves "clean" design history trail of documentation.	Not realistic for more than simple software. First software available late in life cycle. Testing starts late in life cycle. Requires interpretation and modification to make it work for real projects.
<b><i>Modified Waterfall</i></b>	
Intuitive, easily understood. Established and well known. Enforces design before implementation in a slightly more realistic way. Control points clearly demarcated. Leaves "clean" design history trail of documentation and updates.	Still slightly unrealistic ... but better. First software available late in life cycle. Testing starts late in life cycle. Still requires interpretation for changes requiring more than single-phase updates.
<b><i>Sashimi Waterfall</i></b>	
More indicative of how some projects are done (vs. should be done?) Recognizes the need to "peek ahead" in life cycle. Design-before-implement still present but to lesser extent. First software pieces available earlier in life cycle.	Control points less clear. Documentation of design history challenging. Configuration management (synchronization of design documents, code, and test documents) more difficult. Progress metrics and forecasting difficult.
<b><i>Spiral</i></b>	
May be best model for software whose requirements need to be discovered iteratively. Gets prototype software in users' hands most quickly Opportunities for more formal development in later stages of the spiral. Good for user interface intensive software in which user preference is a key driver of requirements. Integration of risk management is explicit and in-line, not supplementary.	Not appropriate for software whose early prototypes could harm users. Tracing evolution of requirements back to user needs may be challenging. Temptation to end the life cycle at the last "good enough" prototype without adequately reviewed design or adequate testing. Metrics for project management, forecasting, and process improvement challenging.

approach may work perfectly for smaller companies with a very narrow product offering in which all software development does break down into nearly the same life cycle model.

2. Different groups could support different life cycle models embedded in their respective quality system processes that are better fits for their specific responsibilities. For example, new embedded software product development, production software acquisition/development, and sustaining engineering in reality work in very different life cycle models. If each group is responsible for a separate quality system process, then each could support its own SDLC life cycle model. The advantages and disadvantages are the same as the first alternative, but because the quality system disciplines are more finely divided along lines of responsibility and not just technical qualifications, the tuning of the life cycle is more often a good fit than the

one size fits all approach above. Larger corporations with diverse product lines (with embedded software, stand-alone software, IT enterprise level software) may similarly standardize on different life cycle models for their different product divisions with the expectation that the life cycle models for each division is best tuned for the product development activities of that division.

3. There is some rationale for allowing each technical discipline's process to support its own life cycle model. To be more explicit, the software developers could support a development life cycle such as the waterfall or spiral models. The life cycles for systems engineering and for the software QA disciplines may look very different from the software development life cycle for the development team. Each discipline could develop its own life cycle model being very careful to understand and coordinate the interfaces between disciplines. This has advantage of being tuned by discipline and the same disadvantage that not every project will be a perfect fit. Furthermore, interfacing different life cycles within the same project can be very challenging and increases in complexity with the number of life cycles in use.
4. The final alternative to be discussed here is actually suggested by the IEEE Standard for Developing Software Life Cycle Processes [1]. Using the concept of the IEEE Standard applied to the quality system modeling described in the previous chapter (i.e., the four Ps: policy, process, procedure and plan), this approach would work as shown in Figure 5.8 and described as follows:



**Figure 5.8** Fitting life cycle planning into the quality system.

- Divide the quality system into processes along discipline boundaries.
- Define the total activities that are the responsibility of the process for every project they will encounter. This may be a superset as not all activities may be applicable, but the activities need to be considered for each project.
- The project plan for a new project is where the life cycle model is selected. A collection of SDLC models can exist within a quality system, and the plan either selects the best life cycle model for the project, creates a new life cycle model tailored specifically for the project, or experiments with new life cycle models to improve the process. Note that this is the same concept that was presented in the previous chapter in the discussion of procedures. A quality system should allow for a collection of procedures for the same activity so that they tailored for the specific needs of a project. The advantage of this alternative is that the SDLC model specified in a plan can be selected to most closely match the circumstances of the project. This does put an increased burden on the creation of the project plan, but over time an organization should begin to recognize some similarities in the projects they undertake, and old plans can be used as a basis for new plans. The SDLC models can be totally defined within the plan, or the models can be defined in a separate collection of documents that can be referred to in plans in the future.

## The ANSI/AAMI/IEC 62304:2006 Standard

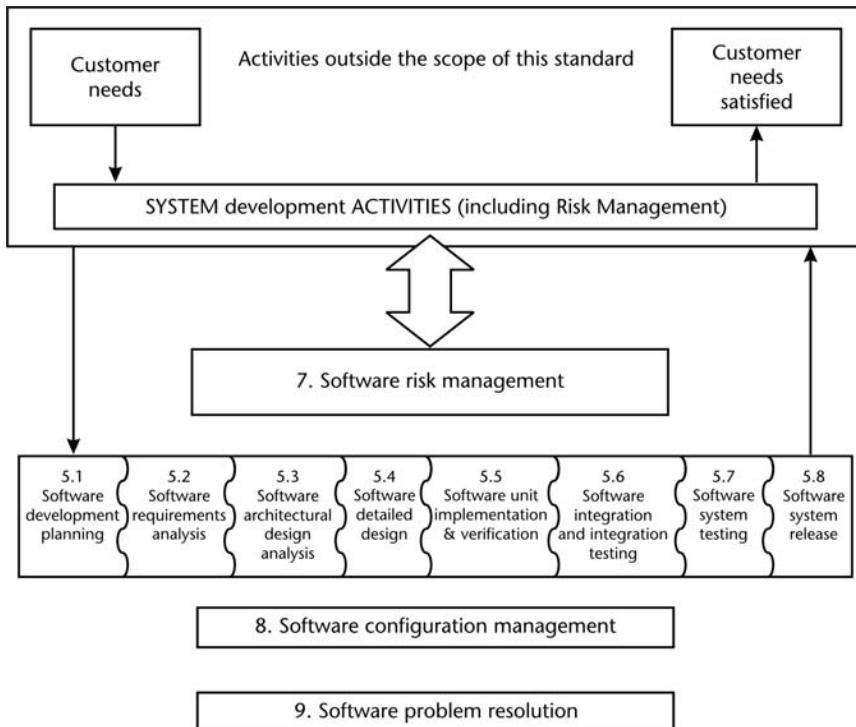
This standard, which is titled “Medical Device Software-Software Life Cycle Processes,” is an FDA recognized standard. The Standard does not actually address life cycles or life cycle models themselves; rather, it defines processes, which are comprised of activities that should be undertaken (or at least considered) for each software development project. Unfortunately, the standard does not use the term “process” exactly the same way we use the term in our discussion of quality systems.

Figure 5.9 shows the organization of the 62304 Standard. The boxes labeled 5.1 through 5.8 (which refer to section numbers in the standard) are the main activities of the software development process that must be included in any software development life cycle or life cycle model. The larger bars labeled 7, 8, and 9 represent supporting processes to the main software development process, which are beyond the scope of the strict development process. The standard is an excellent source for defining the activities to be considered when setting up a quality system for software development and validation.

Just to be clear, a standard like this is not a substitution for one’s quality system; rather, one’s quality system can be designed to incorporate the requirements of such standards to claim compliance.

## An Organization for the Remainder of This Book

Just as a life cycle model cannot be perfect for every software system or development project, no organization of the book can be perfect for every reader coming



**Figure 5.9** Software development processes and activities (from the 62304 Standard).

from whatever background here she is coming from. We are faced at this point with the decision of how to organize the rest of this book. It could be a book that discusses each technical topic and deliverable within its own cocoon. However, that would convey an unrealistic simplicity that does not exist in the real world. Every activity undertaken and every design output produced by the development team has an impact on the validation team. The converse is also true.

It seems the most realistic approach to discussing the topic of medical device software validation is to try to undertake it in the way it would be experienced in the real world of medical device design, development, and validation. To that end, we are faced with selecting a life cycle model around which to organize the text.

The traditional waterfall life cycle model has been chosen as an organizational structure for this text. That is not to say that the traditional waterfall is recommended or preferred. Indeed, it has already been stated that the traditional waterfall life cycle model is unrealistically simplistic for most applications. So, in an attempt not to make a complicated topic any more complicated than it needs to be, the traditional waterfall has been chosen as the most simplistic and intuitive life cycle model around which to organize this text.

## Reference

- [1] *IEEE Standard for Developing Software Life Cycle Processes*, IEEE Std 1074-1997, IEEE Standards Board, December 1997.

# Verification and Validation: What They Are, What They Are Not

Evidently, there is some confusion or misinformation in the industry about what validation means (at least what it means in regulatory guidance). Almost weekly we receive a call from some medical device company announcing that they are putting the final touches on their new device development, and would like to contract us to “validate it”—hopefully in less than a week! This might be understandable for start-up companies who have not been through the validation and regulatory submission process before. However, we also get similar calls from midsize companies who have been through the process a number of times.

Closely related are calls from prospective clients, who call to ask assistance for device submissions that are rejected because of the software validation records they did or did not submit, or those who call because an FDA inspection of their quality system has turned up problems related to software validation. The device companies making these calls think that a call to experts will result in “touching up” the submission, adding a few tests, or some minor paperwork adjustments that will satisfy the regulators. Unfortunately, this is a clear indication that these companies do not understand that validation activities should have taken place throughout the design and development process. They really do not understand that there is actually some value in the validation process—it is not simply a paperwork exercise to satisfy regulators. Cleaning up after the fact generally is not a pleasant process, nor is it likely to add much value to the quality of the device.

Understanding what is expected from validation early in the design and development process is central to maximizing the value to be gained from validation activities, and is equally important for documenting those validation activities in a way that can be effectively communicated to the regulators in submissions and inspections.

## What Validation is NOT

Let’s start with some of the common misconceptions about what verification and validation are. Figure 6.1 summarizes some of these myths. First and probably most common is that validation is simply a lot of testing at the end of the product development process to ensure there are no defects in the product. This is wrong for two reasons. As we will see, it is clear that validation is equally focused on preventing defects from being designed into the software, and on detecting and correcting those defects that do manifest themselves in the software. A validation program that is focused only on testing misses the opportunity to perfect the design and implementation before handing it over to the test group.

<hr/> Software Validation Myths <hr/>	
1. Validation=Testing	FALSE
2. Verification=Testing	FALSE
3. V&V=Testing	FALSE
4. Verification=Validation	FALSE

**Figure 6.1** Validation misconceptions.

There is a second reason that the “validation = testing” attitude is wrong. A validation program that depends on testing alone for a defect-free device is depending on perfection in testing. Unfortunately, test design, development, and implementation suffer from the same imperfections as software design, development, and implementation. If test engineers exist that have such a high level of attention to detail and are so good at testing that they can find each and every defect in the software, then maybe they should develop the software to begin with!

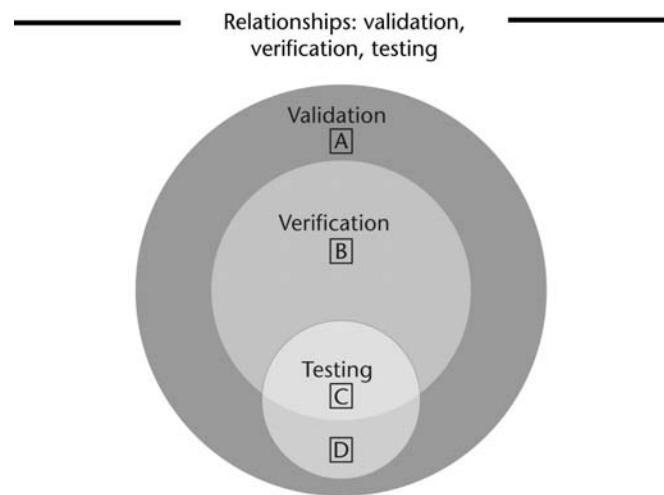
The unfortunate truth is that once humans are involved, imperfection creeps in. The best we can do is set up a system of checks and balances through review and testing of each other’s work to find as many of the problems as we can before the device is used on humans—and even that is imperfect!

OK, if validation is more than testing, then is software verification the same as testing? No. As we will see shortly, verification activities do include testing, but also include reviews and other activities that have nothing to do with testing. Maybe the combination of verification and validation is the same as testing or maybe verification and validation are the same thing? No and No. Let’s try to clarify the relationship between verification, validation, and testing.

## Validation and Its Relationship to Verification and Testing

The Venn diagram in Figure 6.2 clarifies the relationship of these three often confused terms. Validation is represented by the outer ring of the diagram and wholly includes the verification and testing activities, and yet there are validation activities that are neither verification nor testing activities (more on that later). Verification activities are those which verify that individual design inputs have been properly addressed at each phase of the life cycle (more on that later, too). Verification includes some testing. Testing verifies that individual requirements have been implemented correctly. However, reviews are also verification activities, but are not tests (e.g., reviewing design documents to ensure the software requirements are adequately addressed in a design—before code is implemented). While most testing is done to verify implementation of requirements and is part of verification, there are test activities that do not test to verify specific requirements. Examples of non-verification tests include such as user preference testing, ad hoc session-based testing.

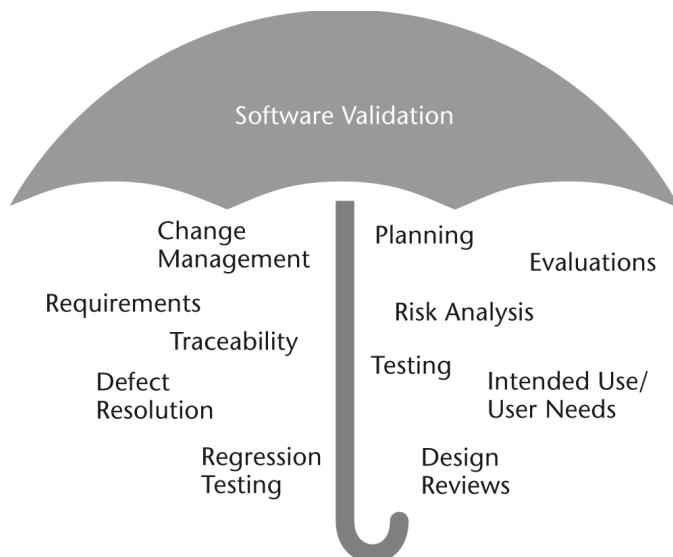
Figure 6.3 shows the collection of most of the activities discussed in the FDA’s General Principles of Software Validation that fall under the validation umbrella.



**Figure 6.2** Verification versus validation versus testing.

It is clear from this diagram that most of the validation activities are *not* testing activities.

Table 6.1 maps the validation activities of Figure 6.3 into the zones of the Venn diagram of Figure 6.2 to further make the point that validation is much more than just testing. Furthermore, most of these activities cannot be adequately accom-



Software testing is one of several verification activities, intended to confirm that software development output meets its input requirements.

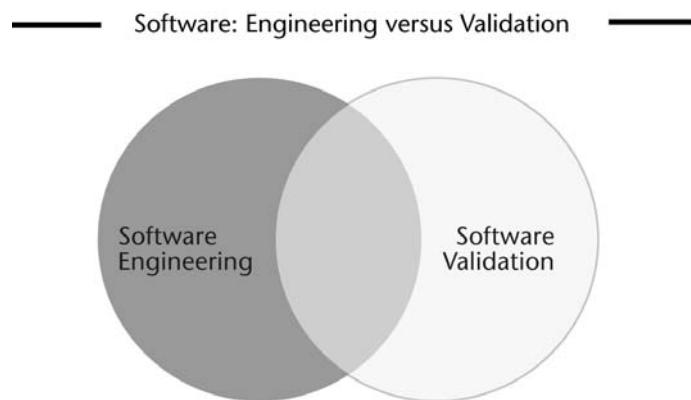
**Figure 6.3** Validation umbrella.

**Table 6.1** Placing Validation Activities in Validation Zones

<i>Validation Activity</i>	<i>Zone</i>
Planning	A
Requirements	A
Traceability	A,B
Change management	A
User site testing	D
Defect resolution	A,B
Risk management	A
Intended use/user needs identification	A
Evaluations	B
Design reviews	B
System, integration, and unit-level software testing	C
Regression testing	C,D

plished only at the end of the development life cycle. Validation must be considered throughout the development life cycle, and the validation activities must be appropriately interleaved with the development activities.

As one starts to parse the definition of validation, and comes to the realization that more than testing is involved, the distinction between validation activities and development activities becomes blurred. In fact, as shown in the Venn diagram of Figure 6.4, much of what is included in software validation is nothing more than good software engineering practices. In topics related to software development life cycles, the role of the organization in validation, and quality system software processes, one might note that much of the discussion is about software development topics. It is precisely because of the intertwined relationship between development and validation that this is so.



The overlap between “good” software engineering methods & practices and software validation activities is considerable.

**Figure 6.4** The relationship between good engineering practices and validation.

## Software Validation According to Regulatory Guidance

Let's review: Software validation is a regulatory requirement specified in 21 CFR 820.30(g) under Design Validation. The regulation does not define what is meant by software validation. The FDA published General Principles of Software Validation (GPSV), which has already been mentioned a number of times in this text. The GPSV expands on the Agency's understanding of what validation means:

Confirmation by examination and provision of objective evidence that software specifications conform to user needs and intended uses, and that the particular requirements implemented through software can be consistently fulfilled [1].

Let's consider the wording carefully and in detail.

"Confirmation by examination and provision of objective evidence" implies that the device manufacturer is obligated to

- Create objective evidence;
- Examine that objective evidence in order to support a conclusion of acceptability of the software based on that evidence alone;
- Provide that evidence to others to convince them of the acceptability of the software.

We'll examine the terminology used shortly, but continuing on with the FDA definition of validation:

Confirmation by examination and provision of objective evidence that software specifications conform to user needs and intended uses, and that the particular requirements implemented through software can be consistently fulfilled.

These underlined clauses are the central theme of software validation; assurance that the software will meet the users' needs and intended uses. The first of these clauses only requires that those device (system) requirements that are to be implemented in software are specified in a way that meets the user needs for the users' intended uses. This is accomplished through reviews, planning, user testing, and other activities. The second clause requires that those software specifications be implemented in a way that meets the specifications consistently. This is accomplished by detailed testing of each specification or requirement of the software. This is also known as verification testing.

In devices of any complexity, analyzing the software in isolation from the rest of the device's subsystems can give a distorted view of the effectiveness of the software. This is true of validation, and as we will see in upcoming chapters, it is also true of risk management, defect management, and configuration management.

In the GPSV, the FDA addresses traceability to system-level requirements:

Since software is usually part of a larger hardware system, the validation of software typically includes evidence that all software requirements have been implemented correctly and completely and are traceable to system requirements [1].

In this part of the discussion of the meaning of software validation, the GPSV addresses traceability from system-level requirements to evidence that the software requirements have actually been implemented completely (every requirement is implemented) and correctly (each implementation works as intended). Note that the reference to the word “evidence” implies that something material is expected for this traceability, whether in handwritten, spreadsheet, database, advanced traceability tool, or other form.

Before moving on to a discussion of verification and its relationship to validation, it will be worthwhile to spend a little time on the terms “evidence” and “objective evidence” that were used in the above two quotations from the GPSV. First, consider that a legal regulatory authority has chosen to use the word evidence in the definition of software validation. It is probably not a coincidence. There are many terms that could have been used (documentation, outputs, results, artifacts, records, files, etc.) It is quite likely that the authors of the GPSV had every expectation that the evidence provided to establish the validated state of device software could in fact also be used as evidence in legal proceedings.

If in fact, there is evidentiary intent behind the use of the word evidence, then the meaning of signatures on documents as described in Chapter 4 takes on an additional level of seriousness. Approved documents, review notes and minutes, test results—basically anything that could be recorded in the design history file should be prepared, signed, and controlled as evidence that could potentially be used in a legal proceeding. Indeed, the Part 11 regulations on the control and validation of electronic records and electronic signature systems exists to assure the veracity, reliability, and security of documents that could be used as evidence.

*Objective evidence.* These two words used together have been the subject of much debate and have certainly cost the industry millions of dollars in effort to gather detailed records that do not really add to the value of the validation effort. In our consulting practice, we often run into software quality or regulatory representatives of our clients who are convinced that objective evidence is a regulation (it's not, it's guidance) mandating the need to gather screen shots or photos of every state the user interface transitions through during a test. Imagine how much time is spent capturing, organizing, labeling, and storing these images. You can be certain that it is more time than was spent looking at the user interface to determine whether the test passed or not!

Unfortunately, somewhere, sometime ago, somebody voiced an opinion that this is what objective evidence means, and those in attendance have religiously followed the advice and spread it to their good friends. This is a classic example of costly activity that adds little if any value to the validation effort. No wonder validation gets a bad rap in a lot of companies!

What is “objective evidence”? As mentioned above, evidence is something material that can be provided to peers, reviewers, and regulatory officials with potential applicability to legal proceedings. Evidence could be test results, photos, or data from test instruments that are a direct output of the instrument or are transcribed by a tester. Evidence is not a feeling, hunch, or a prediction based on past experience. Evidence is not about trust (“We checked that a hundred times trust me it's OK”), hearsay (“Bob told me they checked that a hundred times”), or questioning the abili-

ties of those that created the device (“You don’t need to test that. We know what we are doing, and it would take too long to explain it to you anyway.”).

So, some of you say, those screen shots are evidence! Right. They are evidence. Are they objective evidence? What is meant by “objective”? Objective describes something that is based on fact and not subject to opinion; it can be observed and does not have to be interpreted. So the answer is that those screen shots may be objective or they may not depending on how they are annotated or marked, and how clearly the test procedure instructs the user in the interpretation of the image. From a practical viewpoint, one might consider whether the same level of objectivity could be achieved in some easier way than the time consuming task of collecting screen shots.

Consider the simple exercise on objectivity, shown as Table 6.2. The recorded results of the test are evidence. Note how slight changes in the wording can make the difference between a test procedure that results in objective evidence and one that does not. Also note that the objective evidence is provided by the tester confirming a specific result ... without requiring the collection of a screen-shot. In fact, the screen-shot alone without the objective procedure step would require interpretation and would *not* be objective!

## Can Other Definitions of Validation Be Used?

In Chapter 2 the difference between regulation and guidance was explained and it was stated that guidance only presented one or several acceptable ways of

**Table 6.2 Examples of Procedure Wording that Affects Objectivity of Resulting Evidence**

<i>Example step of a test procedure:</i>	<i>Objective or not objective evidence?</i>
Press the red button. Check the pressure.	Not objective. To a reviewer this is meaningless. What does “check the pressure” mean? Did the tester simply look at a gauge? Did he check that it was safe? Did he have knowledge of what is acceptable (would a tester in the future know what this means).
Press the red button. Check that the pressure is OK.	Could be objective. If “OK” is defined elsewhere and the tester indicates in the results that the test passed, this could be objective (to both the tester and the reviewer)
Press the red button. Record the pressure here: _____	Not objective. This step as it stands alone is not objective. Neither the tester nor the reviewer know if the pressure is acceptable.
Press the red button. Check that the pressure is between 50 and 75 psi.	Objective. Indicates that the expected result is a pressure between 50 and 75. Clear to tester and reviewer that the test passes or fails.
Press the red button. Record the pressure here _____. Check that the pressure is between 50 and 75 psi.	Objective. For all the same reasons above this is clearly objective. The additional gathering of the actual pressure reading does not add to the conclusion (unless we don’t trust the tester to know if a reading is between 50 and 75). Sometimes recording the actual values does help to find other unexpected problems. For instance if the tester writes “it bounces between 25 and 100 but is between 50 and 75 most of the time”. This could lead to subjective interpretations that something is wrong or not.

achieving regulatory intent. In the same chapter some of the flaws with the hierarchy of regulation, guidance, and standards were mentioned. This is one such problem. Clearly, the agency needs to have a common definition of what software validation means in order to enforce the regulatory requirement for software validation. Unfortunately, this definition appears and is explained in a guidance document. So, at least in legal theory, one could validate under one's own definition of software validation (i.e., write your own version of the GPSV). Convincing the regulators that the new definition meets the regulatory intent at least as well as the definition in the guidance could be required. This could result in additional regulatory scrutiny and additional work to explain the validity of the new definitions. For this reason there is a tendency among many (probably a large majority) in the industry to treat the GPSV like it is a regulatory document, not a guidance document.

The discussion of objective evidence and software validation suffer from this same ambiguity. The term objective evidence first surfaces in the GPSV guidance document, but the establishment of a requirement for provision of evidence (in the legal meaning) certainly has the sound of a regulatory requirement.

## User Needs and Intended Uses

These terms “user needs” and “intended uses” are rooted in regulation (21 CFR 820.30g):

Design validation shall ensure that devices conform to defined user needs and intended uses ....

The term “intended use” appears dozens of times in the GPSV. To some extent the terms user needs and intended uses are self-evident. However, it is worth stating explicitly that the regulatory requirement to validate for user needs and intended uses imply a requirement to define those needs and to formally state the intended use(s). Note that the needs and intended uses may be distinct and different from each other. For example, a device’s intended use may be to provide a measurement of blood glucose. The users’ needs may be related to performance (accuracy, precision, response time), sample handling, user interface needs (languages, input devices, complexity), and external interface requirements. The intended use should include the intended users (patients, physicians, care givers, volunteers, expected languages, expected skill level, etc.), intended use environment (ICU, operating room, home, primary care office, etc.), and anything else that may be descriptive of how the device will be used.

## Software Verification According to Regulatory Guidance

The GPSV also introduces the concept of software verification.

Design verification is also rooted in the design control regulations (21 CFR 820.30f in this case). However, in the case of design verification, software is not singled out as it is in case of design validation. That is probably just an oversight (if such a thing exists in regulations). Certainly the GPSV makes it abundantly clear

that software verification is, indeed, part of software validation. According to the FDA in the GPSV:

Software verification provides objective evidence that the design outputs of a particular phase of the software development life cycle meet all of the specified requirements for that phase [1].

Let's parse this paragraph, because each word is loaded with meaning. We are now familiar with "objective evidence"—material artifacts of the verification process treated as seriously as legal evidence. The design outputs are those described at the beginning of Chapter 5. They are the outputs or deliverables that are defined for each phase of the software development life cycle, regardless which model is used. Since we have mentioned life cycles, it is clear from this definition that there is an implicit expectation in the GPSV that software will have evolved through an identified life cycle with specifically identifiable phases. That the "design outputs...of a ...phase...meet all the specified requirements for that phase" is easily put in the context of our discussion on life cycles in Chapter 5. A life cycle phase takes inputs to the phase, acts based on those inputs to create the outputs of the phase. For example in the design phase, an input is the software requirements specification (SRS), the activity is to design the software to meet those requirements, and the output is the software design description. Verifying that the design outputs meet the specified requirements of the phase in the context of this output for this phase simply means to verify that the software design completely and correctly incorporates all the requirements of the SRS. This is handled through reviews and traceability analysis (i.e., verification activities).

The GPSV further states:

A conclusion that software is validated is highly dependent upon ... [the] verification tasks performed at each stage of the software development life cycle.

There are two points to be taken from this quotation. First, that validation is dependent upon verification, thus verification is part of validation as in our diagram of Figure 6.2. The second, and less subtle, point is there is an expectation of verification activities at each phase of the life cycle. This should make it clear that verification, and by inference validation, are not one-time activities that take place at the end of the design and development program. There is an expectation in the GPSV that verification and validation activities take place throughout the development process, and even into design transfer to manufacturing and the maintenance phase once the software is released for clinical use.

To add a little more definition and clarity to the expectations for verification activities, the GPSV further elaborates:

Software verification looks for consistency, completeness, and correctness of the software and its supporting documentation, as it is being developed provides support for a subsequent conclusion that software is validated [1].

This paragraph restates the ongoing nature of verification, reconfirms that verification is a subset of validation, and that verification activities assure the complete-

ness and correctness of the software and supporting documentation. In other words, the software must work like the design documentation says it will; it must be designed to fulfill the requirements specification; and the requirements specification must meet the needs of the system requirements, which ultimately meet the user needs and intended uses. It is not acceptable to allow the design or implementation to evolve or migrate away from what the predecessor documents specify if those changes result in a device that deviates from the user needs. The software and supporting documentation must remain consistent in their description of the device. If the software implementation cannot possibly or reasonably meet the requirements and design specifications, then those predecessor documents must change to represent a solution that can be implemented. The reviews of those changes must assure that the user needs and intended uses are still fulfilled. In other words, software should not simply be changed for the sake of convenience of the developer. There are many other stakeholders (including the intended users) that must be considered.

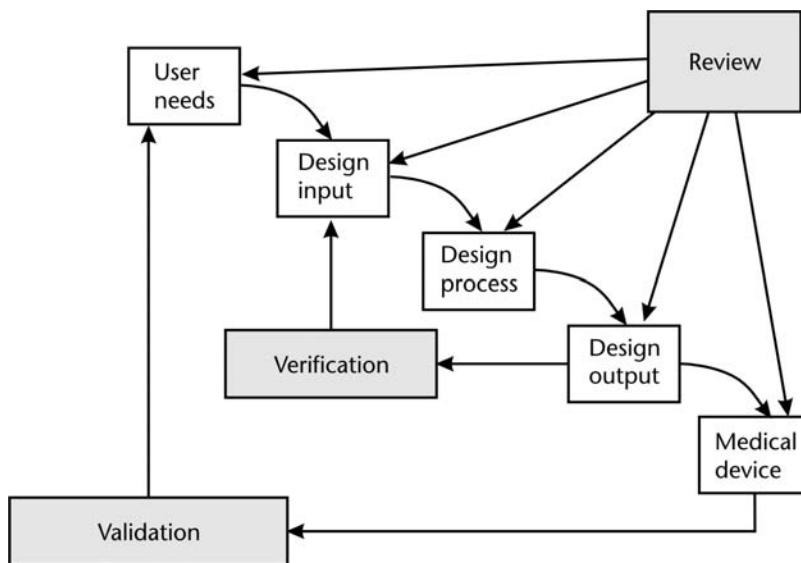
Why? Because unless there is consistency in the software and supporting documentation, it is impossible to make an objective analysis as to whether the software meets the user needs and intended uses. We are highly dependent on that traceability from user needs and intended uses to the software itself by way of the supporting documentation to conclude that the user needs are being met. That is the regulatory rationale. At a more pragmatic level, reviewers and testers cannot come to an objective opinion about the completeness and correctness of a design output unless the inputs have been updated to be consistent with the current state of the design output.

## How Design Controls, Verification, and Validation Are Related

Figure 6.5 is a slightly modified version of a figure in the FDA's Design Control Guidance for Medical Device Manufacturers (1997), which itself was co-opted from the Medical Devices Bureau, Health Canada. As with anything that fits on a single page that attempts to describe the software development and validation processes, this figure too is greatly simplified to make a point.

Figure 6.5 shows the design inputs, design outputs, and design activities for each phase of the software development life cycle. Review points are shown at the end of each phase, and verification activities that include reviews, evaluations, or actual tests are graphically shown to verify that the design outputs are consistent with the requirements or other design inputs of that phase. Of course, this figure is simplified to eliminate the complicating realities of our inability to transition in a one-way fashion through a life cycle. At the top of this waterfall-like model, are the user needs and intended uses; at the bottom is the actual medical device itself.

The “verification” box represents a number of verification activities throughout the life cycle. There is also an outer validation loop that points from the resulting device back to the user needs and intended uses. Why? What does that represent, and why is it needed? The verification activities are meant to keep the design process accurately representing the original user needs as the process progresses from phase to phase, but sometimes, well, things happen. If you ever played the telephone party game when you were young, you will understand the need for the outer validation loop. In the party game a phrase is whispered from person to person sitting around a



**Figure 6.5** Design controls, verification and validation.

table. Inevitably, when the phrase makes it around the table and it returns to the originator of the phrase, something has changed. So it is with the design and development process that can take years to complete. That is why we need a cross-check at the end of the development cycle to confirm that the device that was ultimately developed does, in fact, satisfy the user needs and intended uses. Of course, the validation loop can also represent all of the other nonverification activities that take place during the validation process such as planning, configuration management, and so forth.

It often has been said that verification activities ensure that the device was implemented correctly and validation activities ensure that the right device was developed. It is quite possible to develop a device that passes all its reviews, evaluations, and verification tests, but fails miserably when it gets to the field because it is a perfectly designed and implemented wrong device. This can be the result of faulty needs analysis right from the beginning, or needs that changed during the course of the development cycle. It can also be the result of unintentional migration of the design or interpretation of user needs during a protracted development cycle. Developing the wrong device can also be indicative of a failed design control system. Regardless, perfect design controls and perfect verification activities are not adequate measures alone to ensure that a device will meet user needs when it is released for clinical use unless the final device is subject to some validation testing for its intended use, in its intended environment, with its intended users.

## Validation Commensurate with Complexity and Risk

The GPSV in a number of contexts refers to validation activities commensurate with the complexity and/or safety risk of the device. In Chapter 8, risk analysis and risk

management will be discussed in some depth, but perhaps this is as good a place as any to take on what validation commensurate with complexity and risk might mean.

Neither the GPSV nor any other regulatory guidance document clarifies exactly what it meant by “commensurate with complexity and risk.” This is one of those terms that we all “kind of know” what it means but it’s doubtful that many of us know exactly what it means.

In our experience, we have seen various clients and organizations try to deal with this in different ways. The ANSI/AAMI/IEC 62304:2006 Standard for medical device software life cycle processes as described in Chapter 5 also calls for devices to be put in risk classes. The activities of the various processes of the life cycle are annotated with whether they are required or not depending on risk classification. A close study of this will reveal that not many activities are actually rejected for the lower risk classifications. The standard starts to run into difficulty when dealing with complex devices with multiple software items that may be of differing levels of safety risk. However, it is an accepted standard way of dealing with validation commensurate with risk.

We have seen and authored some test plans, especially unit-level test plans that define complexity by defining thresholds for the number of lines of code in a unit, or by the presence of conditionals, calculations, or branching in a unit. Anything not above the complexity threshold (which is set based on experience and common sense) is highly unlikely to be found defective in unit-level testing and is excluded from what is mostly a documentation exercise. That deals with the likelihood of error based on complexity, but risk is another issue. If a software unit is part of a high-risk function in the device the plan may call for it to be tested regardless of complexity. Risk or complexity based determinations of what tests (at any level) are run as regression tests, or must be run before final release is yet another form of verification test activity commensurate with complexity and risk.

In practical application, our experience has been that the level of detail of requirement, the amount of review, the quantity of tests written, and the number of times the tests are executed all increase with the level of safety risk of a particular function or feature of the software. Likewise with complexity; the more complex the code the more effort is needed to define its requirements and design, review its design and implementation, and for its testing and other validation activities. This is a natural relationship, and in fact it would be difficult to rationalize reduced activity levels for the larger, more complex, or riskier parts of the code. Consequently, the test plans may not emphasize or categorize risk classifications to prove activity commensurate with complexity and risk. Rather they identify risks and focus attention and activities to build confidence that the probability of the risk occurring is as low as possible through verification and validation activities, or that the severity of a failure of the high-risk code is reduced through various risk control measures.

Yet another approach that we have seen used successfully is to design and compartmentalize the software with controlled interfaces between software subsystems so that some subsystems may be classified as high risk, with others are sufficiently low risk in their functionality and are isolated enough from the rest of the system that they can be classified as low risk and held to a lower validation standard.

The point is that many different approaches can work. Some may be more troublesome (and arbitrary) than others. However, an awareness of complexity and risk

and a willingness to do the right thing will generally lead one to make the right decisions on validating commensurate with complexity and risk.

## Is All Validation Created Equal?

Yes and No.

First, let's think about what kinds of software may be subjected to validation under regulatory oversight. Up to now, we primarily have discussed device software. Device software may be embedded software that controls part of a device, communicates with the outside world (including the user), or provides calculations or analysis of data. It could be stand-alone software that is itself a device such as blood bank software, or a program running on a smartphone to manage diabetes based on user inputs. The software might be an accessory to the main device. Within these types of device software, there may be different sources. Device software may be comprised of custom software written specifically and proprietarily for the device. Increasing amounts of device software is being included from off-the-shelf (OTS) sources. The OTS software may be operating systems (real time or PC-based), database management subsystems, communications software (Internet, WiFi, Bluetooth, etc.), signal processing software, or software for other functionality that is easily available in the public domain. OTS software is available from a number of sources ranging from megacorporations like Microsoft, to small privately held suppliers, to freeware, shareware, or open source software. Many devices have legacy software from ancient predecessor devices that is not documented, and nobody left in the company even remembers how it works or why it is written way it is.

In addition to the device software we most often think of as the regulated software in the medical device industry, there is the software used by the manufacturer in the design, development, production, or in any other control of quality of the device, including electronic records and electronic signatures. This is a very broad category of software that includes everything from spreadsheets (yes, a spreadsheet is software) to very complex enterprise-level IT systems for collection of complaint data from around the world. The variety of types of software in this category also include software from the same sources mentioned above ranging from in-house custom written to OTS “shrink-wrapped” applications that are acquired for use.

So to answer the question of whether all software validation is created equal: Yes, the same general principles laid out in the GPSV will apply to any software when applied with a level of critical thinking and common sense. Not all principles will apply to every type of software, nor will they apply to software from all sources. That's where the critical thinking comes in. That's why this book exists.

## Reference

- [1] *General Principles of Software Validation; Final Guidance for Industry and FDA Staff, Section 3.1.2, January 2002, U.S. Department of Health and Human Services, Food and Drug Administration.*



# The Life Cycle Approach to Software Validation

In Chapter 5, the regulatory basis for software life cycles was covered. The rationale for life cycles as a means of controlling design and validation activities was also covered.

Chapter 5 also distinguished between the overall software life cycle and the software *development* life cycle. A number of widely accepted models for development life cycles were presented, compared, and contrasted. In this chapter, life cycles and validation will be examined in more detail. Frequently, validation activities are hung onto a development life cycle like mismatched appendages. In fact, all too often companies standardize on one corporate software development life cycle for all products, practices, and projects. Unfortunately, it is rare that a company's products and projects are so nearly identical that a single life cycle model is sufficient. Forcing a team to use a mismatched life cycle model can lead to confusion and busy work to document how a misfit life cycle was distorted to fit. Oftentimes, it really is not used, or the activities are loosely interpreted and contorted simply to fit into the "standard" life cycle. This kind of retro, forced documentation does little to boost one's confidence in the software. Worse, it erodes the confidence of the development team, test team, and QA team in the process. The erosion of confidence leads to the smartest contributors spending their time looking for ways around the system rather than being the strongest advocates of a system that repeatedly and reliably produces high quality software.

Why life cycles again? There are three main advantages to managing software (or any) projects to a defined life cycle:

- Project planning;
- Design control;
- Project management.

Life cycles are a good planning tool. One cannot identify a life cycle without thinking through how the project will be handled. Articulating the phases of the life cycle, and detailing the activities and outputs of each phase of the life cycle is, well, part of the product development and/or validation plan.

In Chapter 5 the point was made that life cycles highlight well-defined control points in a development life cycle. Why is control important? Controlled design, we believe, leads to better software. Changes are not made to the software unless they are first thoughtfully reviewed and approved. The impact on supporting documentation, and the ability to meet user needs is considered in the process. This cuts down on surprises, last-minute changes (which are widely accepted to be defect-laden), and latent functionality (that escapes review and test).

Knowing where one is in a life cycle is at least a crude metric of project progress. Keeping records of how much time is spent in each phase of a project relative to the size of the project provides a long-term means of measuring performance in an organization. Well-defined phases in the life cycle lead to well-defined activities in project planning that can then be tracked during execution.

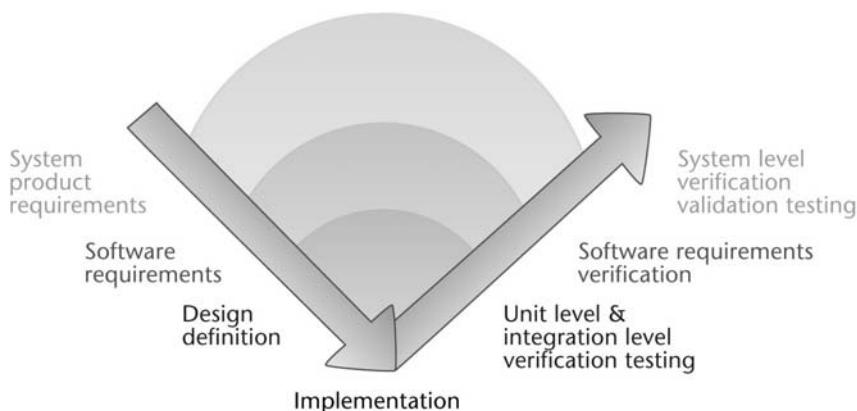
## Validation and Life Cycles

In Chapter 5, most of the discussion was about SDLCs, software *development* life cycles. Certainly good software engineering practices that include planning organized around a life cycle boost our confidence in the software. Consequently, they are considered validation activities in the FDA definition of validation. In this chapter, we examine specifically life cycles and validation activities.

Forsberg and Mooz [1] first proposed the V model to diagrammatically indicate the parallel paths of development and validation. For each development activity, there is a corresponding validation (more correctly, verification) activity. Although this is not a life cycle strictly speaking, and is often confused with a life cycle, it is a basis upon which we can build a validation life cycle model. The V model now exists in many forms as many since 1991 have enhanced and tuned the model for their own purposes. In fact, the IEC 62304 life cycle process standard [2] has its own version of a V model. In Figure 7.1 a version of the V model is recreated for our purposes to show these important concepts.

For example, Figure 7.1 shows the development activity of identification of software requirements coupled with the verification activity of system-level software requirements verification activities.

In Figure 7.2, the waterfall development life cycle is repeated as it was first discussed in Chapter 5. Again, we recognize that the waterfall is usually too simplified to be practical for projects of any reasonable size, but the waterfall is intuitive and best to be used for discussing the concept of life cycles. For the purpose of this



**Figure 7.1** The V model for development and validation.

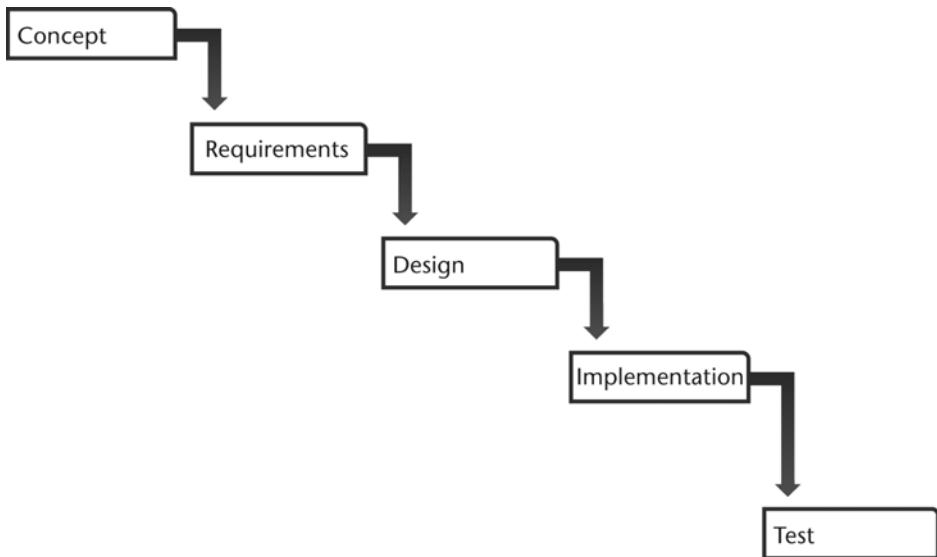


Figure 7.2 Waterfall software development life cycle.

discussion of life cycles and validation, we will assume that the project is one in which the software is being developed from scratch, rather than a modification of an existing version of the product.

The traditional waterfall is usually criticized for being too simple because it ignores the iterative nature of software development. In the context of this chapter, however, this simple life cycle model will be examined to represent the relationship between development activities and validation activities.

## Combined Development and Validation Waterfall Life Cycle Model

Those who prefer to standardize on the traditional waterfall model because of its simplicity usually account for the development and validation activities as shown in Figure 7.3. In this model, the development and validation activities are simply mixed into the traditional phases. In this figure, each phase, which is named for the development activities that primarily take place in that phase, actually contains activities for testing, quality assurance, and regulatory compliance in addition to development.

In this figure, the validation activities may be part of the development, test, quality assurance, or regulatory responsibilities. This approach is used by many. However, just as the traditional waterfall model is overly simplified for representing the development activities, the oversimplification is extended into oversimplifying the relationship between the development and validation activities. If one tries to manage a project into neatly defined phases as shown in this model with validation activities included, one quickly finds that the validation activities often run over from one phase to the next. As an example, where does one put the development of the system-level software test procedures? Clearly, one needs software requirements to write these tests, so they could be developed in the design phase that follows the

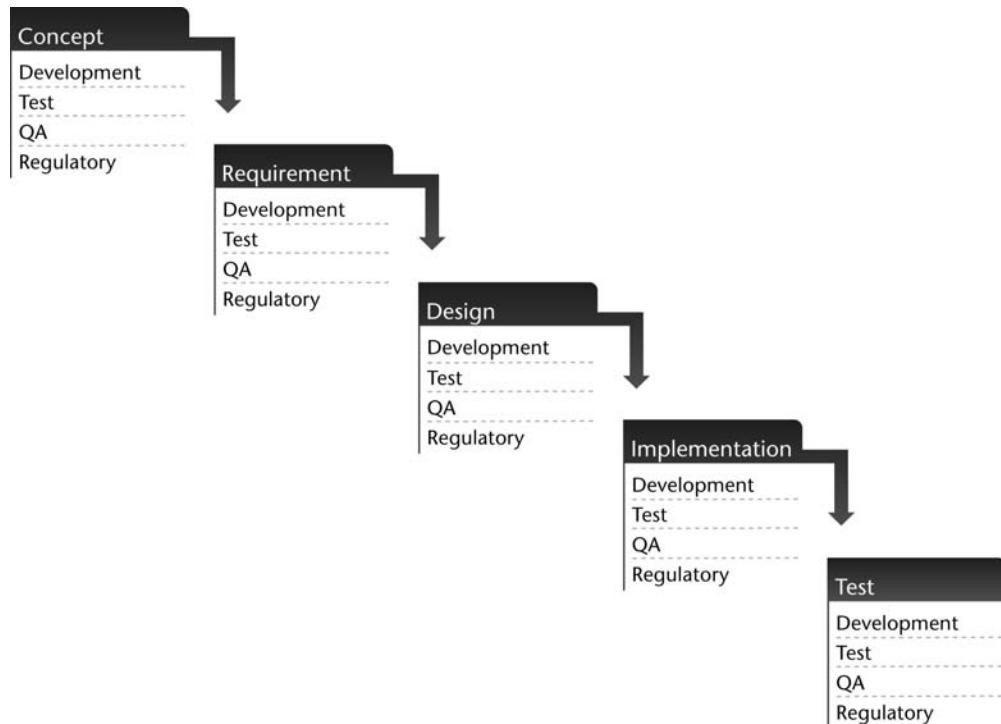


Figure 7.3 Combined development and validation life cycles.

requirements phase. However, the length of time needed to develop the system level test procedures may well be longer than the time needed by the development activities in the design phase. Would one want to hold up the completion of the design phase for the completion of the system-level software test procedure development? Probably not, so the system-level software test procedure development activity bleeds over from the design phase into the implementation phase. There is really nothing wrong with that, but it does eliminate an opportunity that could be afforded by phase driven project management to hold the test development team accountable for completion of a milestone (i.e., completion of the procedures) before the phases complete. The simple point is that the milestones for the validation/test team simply do not align neatly on the phase boundaries that are set by the development activities. In this scenario, the product development/validation team will always be in a state of wondering exactly which phase they are in. Unfortunately, all too often, the answer is that they are a little bit in all phases. When that happens, the usefulness of the life cycle model is almost entirely eliminated.

The approach of Figure 7.3 does not account for opportunities for conducting parallel activities. In the example used above, a well-managed requirements development process will freeze the requirements in the earlier completed and less volatile sections of the requirements document while later sections or more difficult and volatile sections are still being worked out. This does afford the opportunity for the test development team to begin the test development process for those first completed sections. Understand that there is some risk that as the requirements development process proceeds, some changes may be necessary to those previously frozen sec-

tions. Nonetheless this is commonly done to try to advance the product development schedule. Representing that type of project management style would then spread the test development activity across three phases (requirements, design, and possibly implementation). One would be hard pressed to modify the life cycle of Figure 7.3 to account for that reality.

All of these things can be done and are done with a simple traditional waterfall model. The point to be made here is that the traditional waterfall phases are usually selected to represent development activities, and rarely are a good fit for the validation activities.

## A Validation Life Cycle Model

So what can be done? It is somewhat common for the development team and the validation test team to have separate project plans: the development plan and a test (or validation) plan. This makes good project management sense since, most often, there are two different people responsible for the development and test activities. Separate plans allow the two managers to be responsible for the preparation and execution of their plans. If the plans are separate, why can't they support separate but coordinated life cycles?

What might a validation life cycle look like? Anyone who has developed test procedures for any length of time has probably recognized that the steps one goes through to develop a test procedure are basically the same as those that one goes through to develop the software. In Figure 7.4, an example of the validation life cycle is shown. In this model, the first phase of the validation test life cycle is the def-

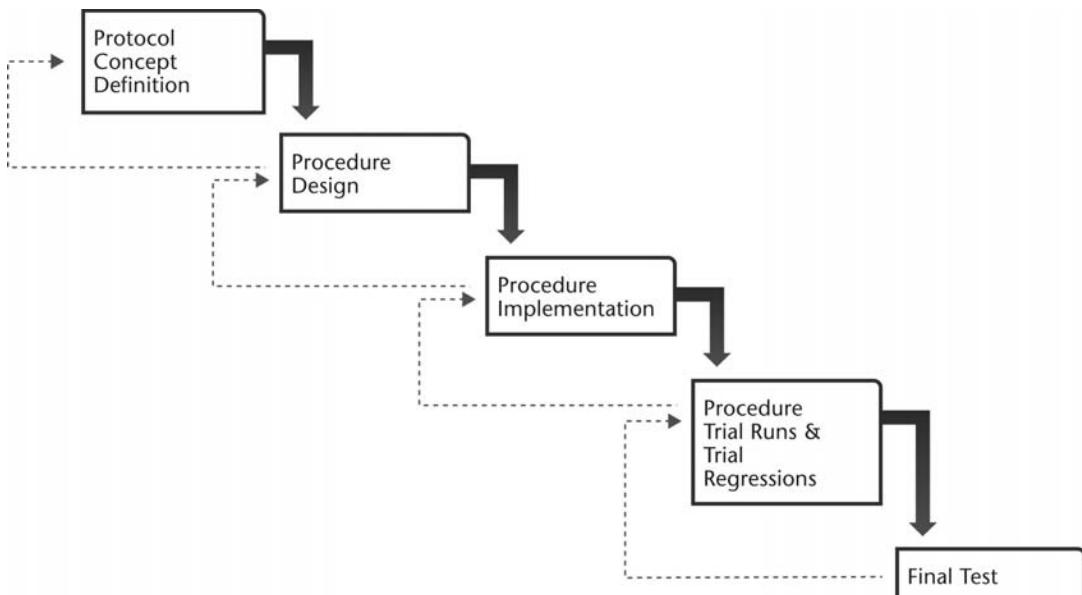


Figure 7.4 A validation test life cycle model.

inition of the groups of test procedures that will be created. Collections of test procedures are often called protocols.

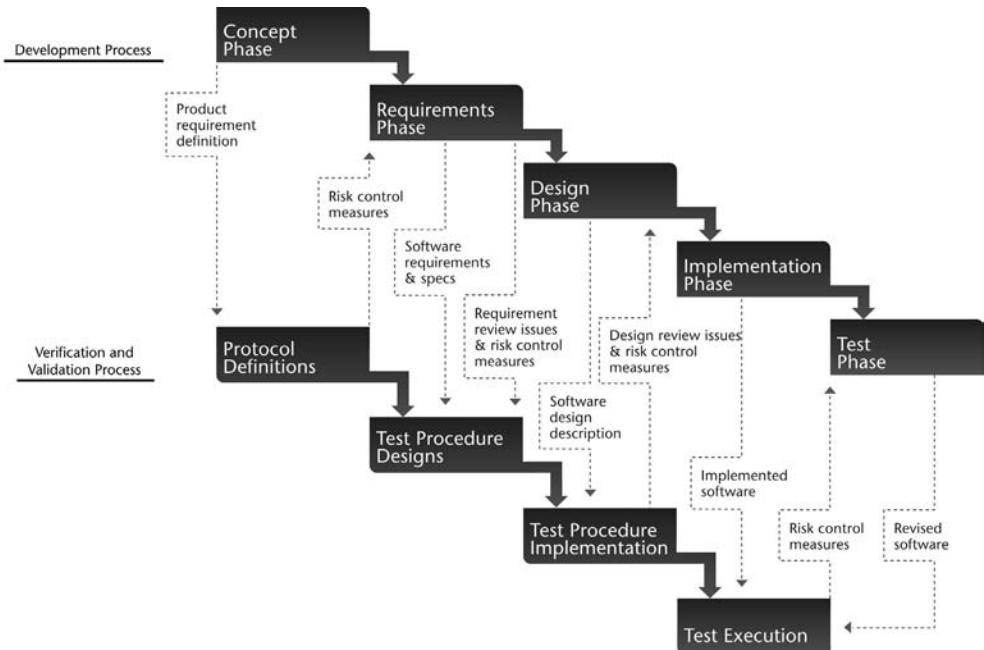
The next phase is the design phase for the procedures. In this phase, the high-level logic and itemization of individual test cases is determined. These designs are reviewed and approved before proceeding into the next implementation phase. The review is important just as the review of software design would be important to keep the test development team from investing valuable time in developing detailed procedures to implement flawed logic. In the implementation phase, the detailed test procedures are written, and again reviewed to be sure that the procedures fully and correctly implement the logic of the procedure design.

Under this model, the reviewed and approved test procedures then move to the next phase in which they are trial run on the implemented software. The validation test team stays in this phase until the procedures are perfected, and the device software is determined to be ready for final test. Test procedures are rerun as necessary when corrections are made to either the test procedure or the software. The last phase is the final test phase. If all has gone well and according to the process, this phase should be little more than a formality.

This sounds pretty simple, right? Well, unfortunately a simple waterfall model for validation test procedure development suffers from the same flaws as the simple waterfall model for software development. The phase transitions are not distinct in the sense that there will not be a single day in which one of these test validation phases is completely finished and the team transitions into the next phase. The more likely scenario is that some designs will be completed before others and will cross over to the next phase for trial run. It is also likely that there will be an iterative component in which test procedures will have to travel backwards through the phases. If a procedure design is found to be flawed during trial run execution, then that procedure will have to be redesigned and reimplemented before it can reenter the trial run phase. Despite the shortcomings, a simple waterfall model for test procedure development is somewhat intuitive, and will get us a long way in the discussion in this text.

Figure 7.5 combines the development and validation life cycles as parallel life cycles, the top being for the software development process, and the lower being for the software validation process. These two processes need to be carefully coordinated since the inputs to the validation process depend heavily on the outputs of the development process. In this example, the validation process has been expanded to include other validation activities than simply the test activities (such as some review and risk management activities).

The size and positions of the boxes that represent the individual phases of the two life cycles are not meant to represent the relative sizes or temporal placement of those phases in relationship to each other. The arrows that go between the two life cycle models represent the dependencies between the two life cycles. For example, the validation life cycle's protocol definition phase needs as an input the product requirements definition of the concept phase of the development life cycle. Similarly, the requirements phase of the development life cycle needs as an input the risk control measures that result from the preliminary risk analysis of the protocol definition phase of the validation life cycle. Note that this is a simplified model with a small



**Figure 7.5** Separate, parallel life cycles for development and validation.

number of dependencies simply to illustrate how two life cycles can be coordinated with each other.

Even though parallel life cycles for the development and validation processes solve some of the problems related to fitting the activity to the life cycle, the simple traditional waterfall models still both suffer from the basic criticism that they do not take into account the iterative nature of the activities that take place within each of the phases and across phase boundaries.

## The Generic or Activity Track Life Cycle Model

Before presenting one last possible approach to life cycle models, let us just review the three main activities to benefit from managing to formal life cycles: project planning, design control, and project metrics. If we choose a life cycle that does not accurately represent the way we perform our development and validation activities, then it is of reduced value in project planning, the control points are not distinct for design control, and the metrics will be meaningless if we are simultaneously working on activities in all phases of the life cycle.

Many of the objections to the models like the traditional waterfall that have well-defined phase boundaries stems from the assumption that activities (like requirements definition) cannot cross over phase boundaries. This likely comes from the names associated with the phase (such as requirements phase). The naming of phases in less well-defined life cycle models such as the sashimi model, the spiral model, or the agile model, are problematic also because they have very vaguely defined boundaries between phases—if boundaries even exist. If one of the objec-



**Figure 7.6** The generic life cycle for activity tracks.

tives of following a life cycle model is to exercise control over the design, then the control points are of prime importance. If one assumes that the best control points are at the phase boundaries, then less well-defined boundaries will result in less well-defined control over the design.

A generic life cycle or activity track approach is shown in Figure 7.6. The differences between this model and others previously discussed are small and subtle, but very important. The activity track phases are named generically. The figure simply names them Phase 1, Phase 2, and so forth. The generic naming sidesteps any confusion caused by trying to apply meaningful names. The activities conducted in each phase are planned, but flexible so that the schedule should not suffer needlessly due to strict adherence to phase definitions. In this approach, there is nothing wrong with iteratively developing requirements, designs, test procedures, or other development/validation artifacts across phase boundaries. The key to controlling the design, while allowing this flexibility is in planning and defining well-defined milestones or “gates” that define the end of a phase.

The milestone definitions are represented in Figure 7.6 by the stop-sign icons between phases inferring that the milestone requirements need to be met before transitioning to the next phase. We will deal with milestones in more detail below. For now, suffice it to say that they are important to this approach because the milestones and the management of their completion (or alteration) is how the design, development, and validation of the device are controlled.

Turning attention to the phases, it should be pointed out that there is nothing magic about the number of phases in a project using this life cycle model. Some projects may have four phases; others may have fourteen or more phases. The size and complexity of the project is related to the number of phases. The intricacy of control needed for a project is the primary determinant of how many phases are required. However, the correct way to think about this is that the complexity of control will determine how many control points (milestones, gates, reviews; or pick your own terminology) will be required. The phases are simply the activities completed between control points; so, the number of phases is really determined by the number of control points defined.

Note that in this approach, it is not necessary to separate the development process life cycle from the validation process life cycle. The life cycle phases are sized according to the activity level necessary to meet a milestone. The milestones are agreed upon in advance and activity is constrained to a single phase at a time. The familiar objection that it is impossible to manage to a life cycle because of the iterative nature of document, test, and software development is addressed by actually planning for multiple iterations of each design output.

Figure 7.7 shows an activity track life cycle definition for a small software development and validation project. Beneath the band that labels the phases are a series of

rows that are the activity tracks. Each track generally represents a collection of activities, although a track could represent a single activity or task. The bars shown in this figure for each track simply show the relative effort required for each track in each phase of the life cycle. It is not necessary to create this diagram for each project. It is prepared for this text to make two important points about this life cycle model:

- Looking vertically at each phase column in the diagram, it would be difficult to put anything other than a generic label on it. Starting in Phase 4 there is activity on all of the activity tracks. It simply would be misleading to call this the design phase or the implementation phase. The overlapping activity tracks afford an opportunity to shorten development schedules by simultaneously working on multiple activity tracks. Unfortunately, that also creates the opportunity for chaos if not controlled by careful planning of the milestones and activities that support the achievement of each milestone.
- Looking horizontally across each row of Figure 7.7 it is evident that almost every activity track has activity in each phase of the project. This represents the predefined iterative releases of each activity output. Some are planned to break a large output into pieces to allow other activity tracks to work with that output. Some of the iterative activities exist simply keep the outputs up to date with the latest versions of their inputs. For example, as risks are better understood in later phases of a project, additional control measure requirements will be defined that must be represented in the requirements, designs, code, and test procedures.

The first three rows of this figure represent the development and iterative evolution of the product requirements definition (PRD), software requirements specification (SRS), and software design definition (SDD) for the software development process. For each of these rows or activity tracks, multiple releases of the output of



**Figure 7.7** Relative activity track effort by phase.

each activity are indicated. Rather than ignoring that each output will evolve throughout the life cycle, this approach actually plans for that evolution. The planned, phased evolution of design and validation outputs focuses the team on completing what is actually achievable in a planned release. At the bottom of the figure are the validation activities. (Note that both the development and validation activities displayed in this figure are a subset of what would be required for most real-world projects.) The validation activity outputs are also evolved across multiple phases.

One might think that the activity track diagram is simply a spiral or agile life cycle that is unraveled and displayed in linear fashion. However, there is a subtle difference. The content of the output of each activity at the end of each phase of the life cycle is planned. Early phases are planned in more detail than later phases simply because it is easier to predict the near future than the distant future. The same is true for plans. It is virtually impossible to plan a project with as much detail in the distant future as one can in the near future. That is why, in Chapter 9 on the subject of planning, the suggestion will be made that plans are meant to be evolved and changed. The same is true for this life cycle model as the project evolves, the inevitable will happen; unforeseen difficulties will arise, task teams will fall behind, some decisions will be slow in coming, therefore the plans must change.

The beauty of the activity track model is that activities can shrink, grow, or be moved between phases without violating the life cycle model, while still providing ample control of the design of the project—as long as adequate planning goes hand-in-hand with evolutionary changes to the life cycle. In fact, the activity track life cycle model and planning are so inseparably intertwined that they are almost indistinguishable. At the beginning of this chapter we started with objectives for life cycle models for validation (or for development for that matter). The first objective was project planning. Perhaps it is a sign that we have met the objective in so far as this life cycle model makes it difficult to distinguish project planning from life cycle planning.

Digging a little deeper, specific detailed activities are defined for each activity track on a phase-by-phase basis in Table 7.1 and continuing into Table 7.2. These tables are representative of a small software project, and while not totally comprehensive, they are a good starting template to use for many software projects. The tables are combined into a large table in Excel format for the DVD that accompanies this text.

Rather than a laborious discussion of each cell of the table, we will settle for a few observations here:

1. Looking at the product requirements activity track (row), notice that it has been planned from the beginning that three major release versions are expected. Easily identified market needs, carryover needs from legacy products, and so forth, are covered in the first version. Other product requirements that may require some user research, or rapid prototyping results will materialize in later versions. The structure of other activity tracks follows the release schedule of the product requirements definition. The details of what is included in each planned release is included in the project plan.

**Table 7.1** Activity Plan: Phases 1–4

Phase Name	Phase 1	Phase 2	Phase 3	Phase 4
Activity/Output				
<b>Development Activities</b>				
Product Requirements	Gather historical needs, market, clinical, standards, regulatory, and legal (IP) needs. Version 1 PRD. Include system level risk control measures RCMs from RM activity.	Add performance, accuracy, throughput requirements. Update with new system level RCMs, resolved issues, corrections. Update with resolutions to issues identified in review. Issue Version 2.	Final call for features, and product requirements. Update for RM additions, resolved issues, corrections. Issue version 3.	Update for RM additions, resolved issues, corrections. Issue Version 3.1
Software Requirements		Create SRS Version 1 from PRD Version 1 Requirements - Update with resolutions to issues identified in review.	Create SRS Version 2 from PRD Version 2 Requirements - Update with new Software RCMs resolutions to issues identified in review.	Create SRS Version 3 from PRD Version 3 Requirements - Update with new Software RCMs resolutions to issues identified in review.
Software Designs			Create SDD Version 1 from SRS Version 1 - Update with resolutions to issues identified in review.	Create SDD Version 2 from SRS Version 2 - Update with resolutions to issues identified in review.
Software Code				Create Release 1 Code from selected, stable sections of SRS/SDD-1. Select release objectives to fit expected time frame of this phase
<b>Cross Functional Activities</b>				
Planning	Overall Project Plan	Revise Project Plan to reflect reality, and to add detail to next phase	Revise Project Plan to reflect reality, and to add detail to next phase	Revise Project Plan to reflect reality, and to add detail to next phase
Reviews	Review Project Plan Review PRD-1 Review CM Plan Rewv. Defect/Issue Mgmt Plan Rewv. System Level Test Plan	Review PRD-2 Review SRS-1	Review PRD-3 Review SRS-2 Review SDD-1 Review Integration Test Plan Review Prod. Valid. Test Plan	Review PRD-3.1 Review SRS-3 Review SDD-2 Review Code-1 Review SLVT-1 Review Test Plan
Risk Management	Preliminary Risk Management	Revise RM based on new detail in PRD-2 and in SRS-1 and RM findings	Revise RM based on new detail in PRD-3, SRS-2, SDD-1 and RM findings.	Revise RM based on new detail in PRD-3.1, SRS-3, SDD-2, code implementation findings, test result findings, defect/issue results and RM findings.
Traces		Trace RCM's to PRD and SRS Trace PRD-1 to SRS-1	Update RCM's to PRD & SRS Update Trace PRD to SRS Trace SRS to SDD Trace SRS to SLVT	Update RCM's to PRD & SRS Update Trace PRD to SRS Update Trace SRS to SDD Update Trace SDD to Code Update Trace SRS to SLVT
Configuration Management	Create CM Plan	Revise plan to reflect reality Manage Change to Requirements	Revise plan to reflect reality Manage Change to Requirements, Designs. Verify Phase Document/Software Configuration Compliance.	Revise plan to reflect reality Manage Change to Requirements, Designs, Tests. Verify Phase Document/Software Configuration Compliance.
Defect Management	Create Defect/Issue Management Plan	Revise plan to reflect reality. Add New ID Issues/Defects Verify/Close Resolutions	Revise plan to reflect reality. Add New ID Issues/Defects Verify/Close Resolutions	Revise plan to reflect reality. Add New ID Issues/Defects Verify/Close Resolutions
<b>V&amp;V Test Activities</b>				
System Level Verification Tests (SLVT)	Create System Level Test Plan	Identify Test Protocol Blocks	Revise Test Plan Test Designs for SRS-1	Revise Test Plan Test Designs for SRS-2 Test Procedures for SRS-1 Test Trial Results for SRS-1
Integration Level Verification Tests (ILVT)			Create Integration Test Plan from SDD-1 High Level Architecture	Revise Integration Test Plan Protocol Blocks from SDD-1
Unit Level Verification Tests (ULVT)				Create Unit Level Test Plan from SDD-1
Product Validation Testing (PVT)			Create Product Validation Test Plan Design Valid. Test Protocol Blocks	Revise Plan as needed Validation Test Designs

2. Dependencies between activity tracks are represented in the table. The details of the dependencies can be represented in the various plans.
3. There is no magic in how much goes into each iterative release of an activity output. Here are some objectives to keep in mind while planning:
  - Each activity track should try to match phase durations with other activity tracks as much as possible so that other tracks are not held up from transitioning to the next phase because of one “long pole in the tent.”

**Table 7.2** Activity Plan: Phases 5–8

Phase Name	Phase 5	Phase 6	Phase 7	Phase 8
Activity/Output				
<b>Development Activities</b>				
<b>Product Requirements</b>	Update for RM additions, resolved issues, corrections, test results. Version 3.2.	Update for RM additions, resolved issues, corrections, test results. Version 3.3 if required.	Update for RM additions, resolved issues, corrections, test results. Version 3.3 if required.	Minor Updates as necessitated by Final Testing Results.
<b>Software Requirements</b>	Update for RM additions, resolved issues, corrections, test results. PRD changes. Version 3.1.	Update for RM additions, resolved issues, corrections, test results. PRD changes. Version 3.2.	Update for RM additions, resolved issues, corrections, test results. PRD changes. Version 3.3.	Minor Updates as necessitated by Final Testing Results.
<b>Software Designs</b>	Create SDD Version 3 from SRS Version 3 - Update with resolutions to issues identified in review.	Update for changes in SRS 3.1 and with resolutions to issues identified in review.	Update for changes in SRS 3.2 and with resolutions to issues identified in review.	Minor Updates as necessitated by Final Testing Results.
<b>Software Code</b>	Create Release 2 Code from selected, stable sections of SRS/SDD-2. Select release objectives to fit expected time frame of this phase	Create Release 3 Code from remaining unimplemented sections of SRS/SDD-3	Revise Code to close defects/issues, and to implement changes to SRS.	Minor Updates as necessitated by Final Testing Results.
<b>Cross Functional Activities</b>				
<b>Planning</b>	Revise Project Plan to reflect reality, and to add detail to next phase	Revise Project Plan to reflect reality, and to add detail to next phase		
<b>Reviews</b>	Review PRD-3.2 Review SRS-3.1 Review SDD-3 Review Code-2 Review SLVT-2 Review ULVT-1 Review ILVT-1 Review PVT-1	Review PRD-3.3 Review SRS-3.2 Review SDD-3.1 Review Codes-3 Review SLVT-3 Review ULVT-2 Review ILVT-2 Review PVT-2	Review Changes to all Documents, Code, Test Procedures	Review Final Versions of all
<b>Risk Management</b>	Revise RM based on new detail in PRD-3.2, SRS-3.1, SDD-3, code implementation findings, test result findings, defect/issue results and RM findings.	Revise RM based on new detail in PRD-3.3, SRS-3.2, SDD-3.1, code implementation findings, test result findings, defect/issue results and RM findings.	Revise RM based on revised documents, code, test findings, defect resolutions.	Revisit RM as Final Test results uncover defects
<b>Traces</b>	Update RCM's to PRD & SRS Update Trace PRD to SRS Update Trace SRS to SDD Update Trace SDD to Code Update Trace SRS to SLVT Trace SDD to ULVT Trace PRD to PVT?	Update RCM's to PRD & SRS Update Trace PRD to SRS Update Trace SRS to SDD Update Trace SDD to Code Update Trace SRS to SLVT Update Trace SDD to ULVT Update Trace PRD to PVT?	Update all traces to reflect changes to product/code structure.	Revise as required by changes made
<b>Configuration Management</b>	Revise plan to reflect reality Manage Change to Requirements, Designs, Tests. Verify Phase Document/Software Configuration Compliance.	Revise plan to reflect reality Manage Change to Requirements, Designs, Tests. Verify Phase Document/Software Configuration Compliance.	Revise plan to reflect reality Manage Change to Requirements, Designs, Tests. Verify Phase Document/Software Configuration Compliance.	Manage Changes Verify Configuration/Version Control
<b>Defect Management</b>	Revise plan to reflect reality Add New ID Issues/Defects Verify/Close Resolutions	Revise plan to reflect reality Add New ID Issues/Defects Verify/Close Resolutions	Revise plan to reflect reality Add New ID Issues/Defects Verify/Close Resolutions Assess Readiness for Final Test	Manage Defect Reporting, Resolution Assignment, Verification, and Closure
<b>V&amp;V Test Activities</b>				
<b>System Level Verification Tests (SLVT)</b>	Revise Test Plan Test Designs for SRS-3 Test Procedures for SRS-2 Test Trial Results for SRS-2	Revise Test Plan Test Designs for SRS-3.1 Test Procedures for SRS-3 Test Trial Results for SRS-3	Revise Test Plan Revise Test Designs for SRS Revise Test Procedures New Test Trial Results	Execute Tests Report Defects Regression Testing Document Results
<b>Integration Level Verification Tests (ILVT)</b>	Revise Integration Test Plan Revise Protocol Blocks Integration Test Designs	Revise Integration Test Plan Revise Protocol Blocks Revise Test Designs Design Test Procedures Trial Test Results	Revise Integration Test Plan Revise Protocol Blocks Revise Test Designs Revise Test Procedures Trial Test Results	Execute Tests Report Defects Regression Testing Document Results
<b>Unit Level Verification Tests (ULVT)</b>	Unit Tests & Results for Code-1	Unit Tests & Results for Code-2	Unit Tests & Results for Code-3	Execute Tests Report Defects Regression Testing Document Results
<b>Product Validation Testing (PVT)</b>	Revise Plan as needed Revise Designs as needed Validation Test Procedures	Revise Plan as needed Revise Designs as needed Revise Procedures as needed Trial Results	Revise Plan as needed Revise Designs as needed Revise Procedures as needed Trial Results	Execute Tests Report Defects Regression Testing Document Results

- Sometimes (always) things don't go according to plan. This approach allows moving tasks between phases, or even splitting them among multiple phases. The key, again, is planning. Activity tracks that are dependent on the outputs of other activity tracks will not want to be held up waiting for those inputs. It is possible to split an output into the completed portion, and

the incomplete or unstable portion. The incomplete portion can be moved to a later phase for completion, while allowing dependent activities to work on the completed portion. Teams must focus on completing, in final form, as much as possible on a task rather than starting many tasks but finishing none. This will grind the overall project to a halt.

- It is also acceptable to create new phases, split phases, or combine phases if it suits the needs of the project plan. Any changes to the originally laid out life cycle that result from moving activities or redefining the phase structure must also be represented in the milestone definitions, and the plans of the individual activity teams. In general, restructuring should be a team decision since changes are likely to affect other activity teams who will need to compensate for changes in their parts of the plan.

The amount of detail that can be put in tabular form documenting the activity track life cycle model can start to look very much like a plan. Working with an activity track table can be an important input to a project plan, or can be a big part of the project plan itself. It is worth noting again the close relationship between life cycle modeling and planning.

Life cycles and planning are of little use if the team doesn't follow the plan—or even attempt to do so. For any of this to make any sense at all, the project team needs to be in one phase at a time, not a little bit in all phases at the same time. The rigidity of the traditional waterfall model made it difficult to be in a single phase, and consequently resulted in much criticism of that life cycle model. Even the activity track model, given all of its flexibility, can suffer from the same problem. What is different about activity track life cycle methodology is that the phase definitions can be managed to fit the reality of the project rather than trying to force fit the project team into phase definitions that no longer make sense.

Figure 7.8 is an example of a milestone tracking chart that could be used for managing a project to an activity track life cycle model. Each activity or task that was laid out as an objective for a specific phase is listed as a row in this milestone tracking chart. The first column to the right of the activity name allows the manager to record whether that activity was actually completed in the phase. Sometimes it is just not practical to hold a team back from transitioning to the next phase simply because one or two activities are not complete. To account for this reality, the activity track life cycle model allows the project to transition to the next phase without completion of all of the tasks in a given phase. However, it does call for an accounting on the milestone tracking chart of any task or activity that was not completed, and a description of any tasks that are being deferred to a later phase. Naturally, any deferred activities need to be updated on the plans for the later phases of the project. The milestone tracking chart acts as a project management tool for keeping all project teams in sync and provides a good historical record of the project evolution that can be analyzed at a later time for process improvement.

Can the flexibility of the activity track life cycle model be abused? Of course it can, just like any life cycle model. What is somewhat different in this model, is that the tracking records leave a trail of the project team's performance. It is perfectly natural for some tasks to be deferred. It is even quite understandable that additional phases may be defined in midproject to allow the team to move forward and resync

PHASE 1 MILESTONE RECORD	Complete? (Y/N)	Describe Tasks Deferred to Later Phase		
			Plans Revised to Reflect Deferrals? Specify Which	Output Added to Design History File?
Project Plan - Ver.1	Y	N/A	N/A	Y
Project Plan-1 Review	Y	N/A	N/A	Y
Product Requirements Definition - Ver. 1	N	Legal input not available until next month. Applicable standards requirements not complete. Standards not available until next month. Release PRD-1 without legal or standards inputs.	Phase 2 PRD. Activities modified to include legal and standards inputs.	Y
PRD-1 Review	Y	N/A	N/A	Y
Preliminary Risk Management	Y	N/A	N/A	Y
Configuration Management Plan - Ver. 1	N	Need decisions on internal vs. outsource development to complete. Release CMP-1 without resource information.	Phase 2 CMP Activities modified to note completion of CMP once resource determined.	Y
CMP-1 Review	Y	N/A	N/A	Y
Defect/Issue Tracking Plan - Ver. 1	Y	N/A	N/A	Y
D/ITP-1 Review	Y	N/A	N/A	Y
System-Level Software Test Plan-Ver. 1	Y	N/A	N/A	Y
SLSTP-1 Review	N	Vacation schedules prevented review from being scheduled.	SLSTP-1. Added to Review Activities for Phase 2.	N

Approval to Initiate Phase 2 Activities: \_\_\_\_\_

**Figure 7.8** Milestone tracking chart.

in a newly defined intermediate phase. Projects that just keep pushing forward unfinished tasks or defining large numbers of new phases will easily be detected as projects that run with less control.

## Life Cycles and Industry Standards

One might ask whether industry standards exist for standardized life cycles. IEC 62304:2006 (Medical Device Software—Software Life Cycle Processes) is one such standard. Note that this is a standard for software life cycle processes, not specifically for development life cycle processes. The standard defines processes, activities, and tasks. The processes, are major collections of activities. Of particular interest to the topic at hand, one process in the standard is the software development process. That process is divided into eight activities:

1. Planning;
2. Requirements analysis;
3. Architectural design;
4. Detailed design;

5. Unit-level implementation and verification;
6. Software integration and integration testing;
7. Software system testing;
8. Software release.

Each of these activities is broken down into a number of tasks to be addressed for compliance with the standard. Our purpose here is not to review the 62304 standard, but to point out that it is simply the standard for life cycle processes, not a standard that recommends a specific life cycle. Mapping the processes, activities, and tasks onto a selected life cycle is the responsibility of the project team. The selection of a life cycle is one of personal preference, or simply picking a life cycle that makes sense in the context of the project being planned.

## Final Thoughts on Selecting an Appropriate Life Cycle Model

This chapter has covered a lot of territory on life cycle models and validation. Traditional life cycle models were centered around the development activity with validation activities added to it as an afterthought. This chapter offers several alternatives for managing validation activities to a life cycle model. There is no single right answer that fits every company, every product, in every circumstance. Perhaps that is the major point to be taken from this chapter; no single life cycle model is appropriate for every circumstance that will be encountered by a typical medical device manufacturer. Consider the following circumstances that are likely to be encountered by a large number of device manufacturers:

- New product development and validation;
- Emergency release of software to correct errors;
- Planned revision of legacy software to introduce new features;
- Device software that contains off-the-shelf or open source software modules;
- Introduction of an acquired product to the acquirer's quality system.

In each of these cases, the verification and validation activities are likely to be quite different. Consequently, the life cycle model for those activities similarly would be expected to be quite different. Selection of a life cycle model that fits the circumstance allows an organization to work efficiently to a plan centered around that model, rather than struggling to make an ill-fitting model work.

## References

- [1] Forsberg, K., and H. Mooz, "The Relationship of Systems Engineering to the Project Cycle," *First Annual Symposium of the National Council On Systems Engineering (NCOSE)*, October 1991.
- [2] *Medical Device Software—Software Life Cycle Processes*, ANSI/AAMI/IEC 62304, 2006.



# Supporting Activities that Span the Life Cycle: Risk Management

## Introduction to Activities Spanning the Life Cycle

In the last chapter, during the description of the activity track life cycle model, it became apparent that most activities were represented to some extent in most of the “generically named” phases of that life cycle model. In the more traditional life cycle models, many of the mainstream development and validation activities have traditionally been considered to be substantially completed in a phase that is named after a major activity. For example, in the traditional waterfall model, one would likely expect that the requirements would be substantially completed in the requirements phase.

There is another group of activities that has been historically referred to as supporting activities or supporting processes. In fact, the ANSI/AAMI/IEC 62304:2006 standard on software life cycle processes shows at least three of these activities as parallel to the mainstream thread of development activities. These activities are not mainstream in the sense that they do not result in outputs that are in the mainstream flow of design outputs that start with stakeholder needs and finish with completed and validated software. That is not to say that these activities are not important. In fact, one might argue that one of these activities, the risk management activity, is so important that it guides or moderates the activity level for almost all other activities throughout the life cycle. These activities typically are initiated very early in the product development/validation life cycle, and are present in virtually every phase of the life cycle.

The supporting activities to be discussed in this chapter and Chapter 9 that fit this description are:

This chapter:

- Risk management

Chapter 9:

- Planning;
- Reviews;
- Traceability;
- Configuration management;
- Defects/issue management.

Most people will probably agree that these “supporting” activities are very important in the control and quality of the design outputs. However, the supporting

activities are somewhat different from the mainstream activities of requirements identification, design, implementation, test design, and production of test results. The supporting activities, in one view, are modifiers or controllers of these mainstream activities.

Note that these supporting activities in large part describe activities that could be considered development activities. Each, however, is mentioned in the GPSV as a validation activity as well. Does that mean that there is a development and a validation version of each of these activities? There could be, but there does not have to be—at least from a regulatory perspective. It has been mentioned several times already in this text that there is much overlap between validation activities and good development engineering practice activities. Recall that the whole purpose of validation is to boost confidence in the software. Good development engineering practices are more likely to result in error-free and reliable software. Therefore, at least from the perspective of the FDA's GPSV, many of the good development engineering practices are talked about as though they are validation activities.

## Risk Management

This topic is so large and important for medical device software that it really is deserving of a book of its own.

The recent (i.e., within the last five years) literature is full of references to “risk-based” validation. Recently, I attended a symposium at which one of the speakers actually made the comment, “The major objective of risk-based validation is to reduce the amount of testing activity required.” As we will see in the following, reduced test activity certainly is a desirable byproduct of risk management and risk-based validation, but reduced testing alone hardly achieves the stated objectives of a risk-based validation process. Perhaps this a danger of the proliferation of buzz words; we start developing thought around the buzz word and forget the fundamentals behind the buzz.

As we will see, the regulations, guidance documents, and industry standards are heavily peppered with references to risk. In particular, the GPSV has many references to risk, risk analysis, risk management, and risk control. However, one might legitimately ponder whether risk management is part of software validation or whether software validation is simply a risk control measure. This question probably deserves serious debate someday since all of design control, verification, and validation are about reducing risk, but that is a discussion for another time.

Why is risk management important to software validation? First, let us be clear that unless otherwise specified, in the context of medical device software validation, we mean the risk of harm to a patient, user, caregiver, or other bystander. This is often referred to as “safety risk.” There are other risks that could be considered such as project risk, technical risk, and regulatory risk. The techniques for managing other types of risk are largely the same as for safety risk. However, in the medical device industry, management of safety risk is a higher priority than other types of risk management in terms of its regulatory and ethical importance.

Risk management has a relationship with software validation that is unlike any of the other validations processes or activities discussed in this text. As shown in Figure 8.1 and as briefly mentioned above, risk management is considered a validation activity. A systematic risk management process that identifies risks, designs controls for the significant risk, and verifies the effectiveness of those controls enhances our confidence in the software or system. Since validation's goal is to build confidence in the software, it would appear that risk management *is* a validation activity.

On the other hand, as we will see, risk is a combination of severity of harm, and the probability of harm occurring from a failure of the software or system. There are two ways to control risk; one can manage the severity of harm that results from a realized risk, or one can reduce the probability of occurrence (or both). Validation is confidence building, presumably, at least in part by reducing the probability of failure through good design practices and testing. Since validation reduces the probability of a risk—it appears that validation could be considered a risk control measure. None of the other processes or activities discussed in this text has this kind of circular relationship with validation.

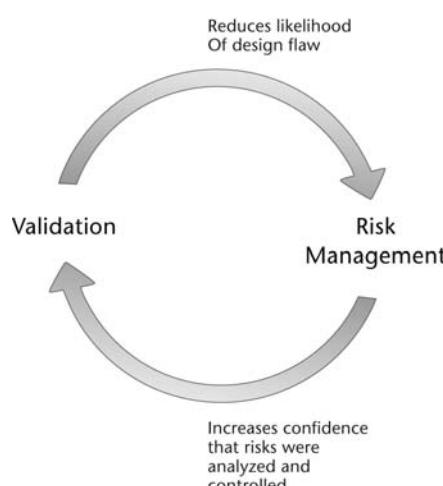
## Risk in the Regulations and Guidance Documents

The term “risk” appears only once in the design control regulations:

Design validation shall include software validation and risk analysis, where appropriate.

—*Design Validation, 21 CFR 820.30-g*

Certainly this is a thought-provoking regulation because it does little to explain what risk analysis is, or under what circumstances it may or may not be appropriate. In fact, come to think of it, when would it not be appropriate? Further, if there



**Figure 8.1** The circular relationship between risk and validation.

are situations in which it is not appropriate, would we not have gone through a risk analysis to come to that determination?

The Design Control Guidance is useful in explaining certain risk-related activities and how they fit into the other design control activities. It also offers a new term, “risk management,” which it defines as:

... the systematic application of management policies, procedures, and practices to the tasks of identifying, analyzing, controlling, and monitoring risk.

*—FDA’s Design Control Guidance for Medical Device Manufacturers, 1997*

The Design Control Guidance also furthers our understanding of risk management by describing it as an ongoing process throughout the development life cycle:

Risk management begins with the development of the design input requirements. As the design evolves, new risks may become evident. To systematically identify and, when necessary, reduce these risks, the risk management process is integrated into the design process. In this way, unacceptable risks can be identified and managed earlier in the design process when changes are easier to make and less costly.

*—FDA’s Design Control Guidance for Medical Device Manufacturers, 1997*

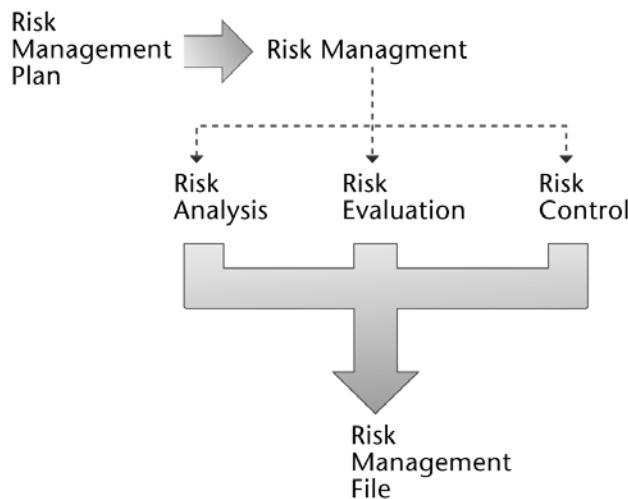
Certainly, it is important to identify risks early to manage the ease and cost of controlling them, but these are seldom goals of the FDA. The real objective is to identify risks early enough so that they are easy and inexpensive to control—so that they *will be* controlled.

## ISO 14971: Application of Risk Management to Medical Devices

In 2000, the industry standard ISO 14971 was released on the application of risk management to medical devices. In 2007, the standard was rereleased in its current form [1]. We will talk about 14971 in some detail in this chapter, but the discussion here is no substitute for actually reading the standard. If there is one must-read document on risk management for medical device, it is 14971.

The 14971 standard introduced the industry to a well defined risk management process comprised of three major activities (analysis, evaluation, and control) and two deliverables (a plan and a file) related to the outputs of risk management activity. The relationship of these terms is depicted in Figure 8.2.

The risk management process is well structured and detailed in 14971. It is an intuitive process that is easy to understand. However, the standard defines risk as “a combination of the probability of occurrence of harm and the severity of that harm.” The problem is with “probability” and software. For mechanical or electrical components or subsystems, there may be detailed specifications (such as failures per million operations, meantime between failures, etc.) available that would allow one to come up with the probability of failure related to defective parts or physical wear on a part. The probability of failure of a physical part is related to the reliability of the part.



**Figure 8.2** Relationship of the components of risk management.

Software, however, does not wear out. If software works for a given set of conditions once, it will always work for that set of conditions. Conversely, if software fails for a given set of conditions it will always fail for that set of conditions. The probability of failure for any given single set of conditions is either zero or one. It fails for that set of conditions or it doesn't—with 100% reliability. An intermittent failure of software is not really the software failing intermittently; it is the intermittent occurrence of a specific input or sequence of events that causes the software to fail—always. It appears to be an intermittent software failure only because the set of input conditions that results in failure has not been fully characterized. That is, not all elements of the failing input condition have been identified. If three of four elements have been identified, the software will not fail until the fourth element is present. If only the three known elements are being monitored, it appears that the software fails intermittently, but it really is just the intermittent appearance of the unmonitored fourth element that causes the software to fail. Regardless of whether the failure of the software is intermittent or not, it is a failure because that set of input conditions was not handled properly, usually because it was not anticipated.

The 14971 standard distinguishes between random failures and systematic failures. Random failures are failures of a component or subsystem due to manufacturing or material variability, or due to wear on the product. Systematic failures are failures of the system that created the device. Systematic failures include design and development system failure. What is unusual about software is that there are no random failures. Any failures that exist were designed into the software (or weren't designed out of the software) and those are systematic failures—a failure of the design and development process.

The intuitive appeal of the 14971 standard still left an open issue for managing risk related to medical device software. The first edition of the 14971 standard in its discussion on systematic failures included:

In cases where an appropriate level of confidence cannot be established for the estimation of systematic failures, the risk should be managed based on the severity of the harm resulting from the hazard. Initially, the risk estimation for systematic faults should be based on the presumption that systematic failure will occur at an unacceptable rate.

There is a relationship between the quality of the development processes used and the possibility of a systematic fault being introduced or remaining undetected. It is often appropriate to determine the required quality of the development process by taking account of the severity of the consequence of the systematic faults and the effect of external risk-control measures.

The second edition of the standard reduced the above to this:

Because risk estimation is difficult in these circumstances, the emphasis should be on the implementation of robust systems to prevent hazardous situations from arising.

Regardless of which wording you prefer, it is clear that there are difficult issues related to risk estimation for software.

## AAMI's TIR32:2004: Medical Device Software Risk Management

In an attempt to add clarity to the meaning of risk management in the context of software validation, the Association for the Advancement of Medical Instrumentation (AAMI) tasked a workgroup of industry and FDA representatives to create an industry Technical Information Report (TIR32:2004 - Medical Device Software Risk Management) on the subject. The report achieved a number of accomplishments in clarifying risk management for medical device software. The report is a survey of risk management techniques used in various industries, and discusses their applicability to the medical device industry. The report examines risk management as defined by the 14971 standard from the software perspective. The report also examines risk management activities throughout the software life cycle, and is filled with insight, discussion, practical methods and information.

TIR32:2004 was also successful in calling attention to the following points:

- A failure of medical device software in isolation cannot alone cause harm to anyone. The harm related to device software failure can only be exerted upon the patient, user, caregiver, or bystander through the system in which the software is embedded. For example, the failure of software embedded in an infusion pump is harmless without the surrounding pump. Even the failure of a software-only device such as blood bank database software is harmless without the surrounding “system” of people, processes, and cross-checks. Consequently, it makes little sense to attempt a risk analysis or hazard analysis of software separate from the rest of the system of which it is a part.
- Risk management for medical devices is more than a one-time activity that occurs early, late, or in the middle of the software development life cycle. There are activities at each phase of the life cycle that relate to risk management. Risk management activities in the early phases of the life cycle achieve

the goal of identifying risk control measures early while they are easy and inexpensive to implement. Risk management activities in the later phases of the life cycle are effective in identifying hazards that could not possibly have been predicted in early phases of the life cycle.

- Risk management cannot be achieved responsibly at the end of the product development life cycle by retro-documenting a risk management process that did not take place.
- Risk management is at least partially an iterative process that repeats itself at every phase of the software life cycle. The results will evolve as additional information about the design and implementation of the software and the other elements of the device are revealed.

## Risk and the IEC 62304 Standard on Life Cycle Processes

The last of the standards to be considered is the ANSI/AAMI/IEC 62304:2006 standard on Medical Device Software—Software Life Cycle Processes. The standard includes a section that defines a risk management process that defines activities specific to risk management and software. Although the risk management process described in 62304 is written as though it is a single one-time process, it is clear from the figures in the standard that it is intended as a process that takes place throughout the software life cycle.

This standard also approaches risk management from a different approach. Three software safety classifications are defined in the standard:

1. Class A—No injury or damage to health is possible;
2. Class B—Nonserious injury is possible;
3. Class C—Death or serious injury is possible.

Each of the development and validation activities described in the life cycle process standard is specified for its applicability to each of the safety classifications. This is an attempt to tune the validation activities to be “commensurate with the level of risk posed by the software” [2].

Although this is an appealing systematic approach to doing more where the risks are higher, the actual implementation of the standard may be somewhat more than one would expect. The first problem comes with the classifications themselves. The standard allows for decomposition of the software into a hierarchy of software items. Child items inherit the safety classification of the parent (unless an alternative segregation rationale is documented). Taken to the extreme, this means that all software that is part of a Class C device will be treated as Class C. If an objective of the standard was to do more for the more risky software and less for the less risky, this classification logic eliminates that objective.

This creates problems on both ends of the risk spectrum. For Class C devices, all software is Class C, and there are no shortcuts for less risky software. For Class A devices, all child software items will initially inherit the Class A safety class. Since the safety classification has to be made early in the life cycle for the software system (since life cycle activities are determined by the safety class), the potential exists that

certain activities may be excluded that might lead to a conclusion that the safety class is higher than Class A. In other words, the safety classification system could itself contribute to safety critical hazards being missed.

For example, some of the testing activities (integration and unit level to name two) are not required for Class A devices. It is not difficult to imagine an integration test that uncovers a potentially unsafe failure mode. However, if the device is declared Class A, the integration test that would uncover the failure mode might never be written or run because the standard would allow lead the developer to skip integration testing for the Class A device.

Does this mean that 62304 should not be used? It definitely does not mean that. It simply means that when using 62304 as a means for establishing what risk-based validation means, one must be aware that blindly following the standard can lead to unintended consequences.

## **IEC/TR 80002-1: Application of 14971 to Medical Device Software**

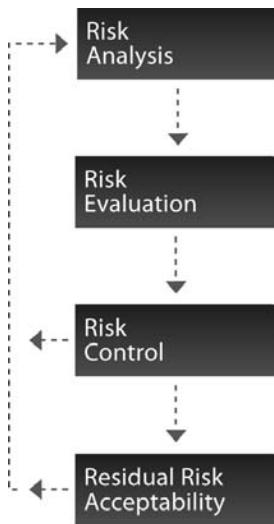
As a transition from discussing the 62304 standard for lifecycle processes to the 14971 standard for risk management for medical devices, a short mention must be made about IEC/TR 80002-1 [3] which was published in 2009. This technical information report authored by an industry workgroup to address the specific needs of applying the 14971 risk management process to devices with specific needs related to software that is part of the device. This report carefully integrates the requirements of the 62304 software lifecycle processes standard with the requirements of the 14971 risk management standard to recommend specific outputs and activities.

This report quotes freely from 14971 and interprets each quotation to a specific interpretation for software driven devices. The annexes offer tables of examples of causes of hazards broken down by software function area, and software causes for that can introduce side-effects. Neither table is meant to be an exhaustive list of all possibilities, but they do represent a good starting point for consideration. Further, they are good examples of the mindset needed to think about one's own specific device's hazards and causes.

## **The Risk Management Process**

The risk management process is described in 14971 in the form of a simplified flowchart, which is reproduced in Figure 8.3 in a somewhat simplified and altered form from that shown in the standard.

As one can see, the process is fairly simple, and quite intuitive. Each of the boxes in the process represents a collection of activities. The risk analysis activities identify individual risks associated with the use, misuse, and abuse of the device. The risk evaluation activities determine whether an identified risk is acceptable according to the definition of acceptable risk in the risk management plan. The risk analysis and risk evaluation activities are sometimes collectively referred to as the risk assessment.



**Figure 8.3** The 14971 risk management process.

The risk control activities identify risk control measures that will reduce the risk's severity, probability, or both. The risk being considered is subjected to Risk analysis and evaluation again to determine whether the risk has been reduced to an acceptable level post-control measure. If not, the cycle repeats.

Risk control measures often add complexity to the device. With added complexity often comes added risk, so risk analysis after application of risk control measures must consider whether additional risks have in fact been introduced.

Once each individual risk has been analyzed, evaluated, and controlled as needed, the overall risk posed by the aggregate of the individual risks must be evaluated for acceptability.

Although not represented in this figure, the standard makes it clear that risk management is not complete when product design and development is complete. The manufacturer has responsibilities related to the management of risk of the device even after it has left the shipping dock.

## The Language of Risk Management

The 14971 standard introduced a number of terms, each of which has a specific meaning. Unfortunately, many of these terms get misused in practice or are incorrectly used as though they were synonymous. Without a clear understanding of the terminology, it is difficult to clearly understand the intent behind the standard. Since we will rely heavily on the 14971 standard, it is worth spending some time on the terminology.

The terminology breaks down into two categories. The categories are:

1. The risk management outputs (sometimes called artifacts);
2. The risk management concepts and definitions.

Let's take a look at the terms and their relationships to each other, then take a closer look at the activities suggested by the risk management process diagram of Figure 8.3.

## Risk Management Outputs

The risk management outputs defined by the 14971 standard are the risk management plan and the risk management file.

### The Risk Management Plan

Although it may seem obvious, the risk management plan is exactly that, a plan. It is a plan just like development or test plans. Risk management plans include roles, responsibilities, resources, and activities related to the risk management function. Frequently, companies confuse the risk management plan with the results of the risk management process (i.e., the risk management file discussed below). Risk management planning can be intertwined with plans for other disciplines related to product development such as a software development plan, or a software validation plan, and in fact this is often done. However, for a project of any size or complexity in which there is any significant risk related to harm to the patient, caregiver, or bystanders it may be valuable to consider a separate risk management plan. For even moderately complex devices, it is difficult to get the big picture of whether risk is being adequately managed on the device level unless at least the risk management plan is pulled together in one place.

One final component of the risk management plan that is slightly different from other plans is the definition of acceptable risk. Risk Evaluation is part of the risk management process. The evaluation of a risk involves determining whether that risk is acceptable, or whether it needs to be reduced further to an acceptable level. The threshold of acceptability needs to be defined in the risk management plan so that all risks are evaluated consistently. The standard provides several methods for defining acceptable risk that are interesting, but are too detailed for this text.

Defining acceptable risk is perhaps the most difficult component of the risk management plan. It is not unusual for the definition of acceptable risk to evolve as the project evolves. Usually this is because the definition is unreasonably ambitious, or vague early in the project. As the realities of risk management activities are encountered, the definition of acceptable risk is often relaxed to a more reasonably achievable level. Of course, the level of acceptable risk could also be made more restrictive as the project evolves and the risk management team is made aware of risks that were not anticipated early in the project.

As discussed in Chapter 9 on the topic of planning, a plan is only applicable until it is determined that it needs to be revised to meet reality. That also applies to the definition of acceptable risk. There is nothing sacred about this definition in the plan that would prohibit it from being changed later in the project. Could this be abused? Of course it could, just as one could abuse any other plan by simply modifying it to compensate for objectives that may not have been taken seriously. If the definition of acceptable risk has to be changed later in the project, at least the risk

management file will be left with documentation of the level of acceptable risk the device was designed to achieve.

### The Risk Management File

If the results of the risk management process are not recorded in the risk management plan, then where are they recorded? The answer is in the risk management file. Risk analysis and risk management do not result from a single technique or tool, nor do they happen in a single meeting at some arbitrary point in the development life cycle. That implies that the outputs of the risk management activities may be varied from project to project. Does that mean the outputs need to be collected in one large physical or electronic file? No. The 14971 standard notes that for the risk management file:

The records and other documents that make up the risk management file can form part of other documents and files required, for example, by a manufacturer's quality management system. The risk management file need not physically contain all the records and other documents; however, it should contain at least references or pointers to all required documentation. The manufacturer should be able to assemble the information referenced in the risk management file in a timely fashion.

What goes in a risk management file? The standard specifies that in addition to any other records documenting the activities associated with risk management, the following items are to be included:

- The risk analysis;
- The risk evaluation;
- The implementation and verification of risk control measures;
- The assessment of acceptability of any residual risks.

## Risk Management Concepts and Definitions

Before digging into the details of the risk management activities, let us get comfortable with the language and terminology supported by the 14971 standard.

Some of the key definitions taken from the 14971 standard are:

*Risk:* Combination of the probability of occurrence of harm and the severity of that harm.

*Severity:* Measure of the possible consequences of the hazard.

*Probability:* (Interestingly not defined in 14971, but it is a widely understood technical concept.)

*Harm:* Physical injury or damage to the health of people, or damage to property or the environment.

*Hazardous situation:* Circumstance in which people, property, or the environment are exposed to one or more hazard(s).

*Hazard:* A potential source of harm.

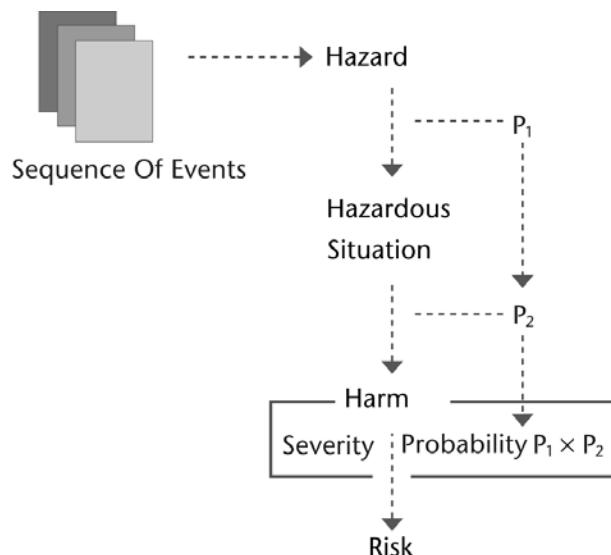
Now, let us look at how these concepts are interrelated in Figure 8.4. A hazard exists in a given system, but is “harmless” unless a sequence of events creates a hazardous situation. That sequence of events may occur with a probability of  $P_1$  and may or may not result in harm. The probability that the hazardous situation does result in harm is  $P_2$ . The harm when it does occur, occurs with a given severity and probability equal to  $P_1 \times P_2$ . That severity and total probability defines the risk that is to be managed.

Adding specific examples often help to clarify the concepts. Annex E of 14971 has the standard tables of examples for each of these concepts if you feel you need more than the one provided here.

#### *Example: Infusion Pump*

Imagine a failure of an infusion pump that is initiated by an unexpected sequence of inputs from an inexperienced user. The indecisive user alternates between input fields on the programming screen a number of times, entering, erasing, and changing the input values. The software has not anticipated this sequence of user errors and resulting inputs. The result is that the volume limit setting shows a valid value on the screen, but the software has calculated a corrupted value because the input routine becomes “confused” by the erratic sequence of inputs. When the pump reaches the volume limit at which it should shut down, it compares the actual volume to the calculated volume limit setting which is orders of magnitude too high. The pump continues to run past the volume limit. The hazard analysis for this simple example is summarized in Table 8.1.

So far, it looks pretty simple, right? This is about as far as 14971 takes the hazard analysis discussion in Annex E. We will revisit this example below in further discussion. Reserve judgment about how simple this example is until the risks are analyzed in more detail.



**Figure 8.4** Diagrammatic progression from hazard to risk.

**Table 8.1** Example Hazard Analysis for a Single Hazard

A	Hazard	Function: Pump won't stop
B	Sequence of events	<ol style="list-style-type: none"> <li>1. Unexpected user programming sequence</li> <li>2. Limit_Value corrupted</li> <li>3. Volume_Limit_Shut_Off function fails</li> <li>4. Pump runs beyond programmed limit</li> </ol>
C	Hazardous situation	Over delivery of infused drug
D	Harm	<ol style="list-style-type: none"> <li>1. Internal discomfort</li> <li>2. Minor (recoverable) damage to tissue or organs</li> <li>3. Major (nonrecoverable) damage to tissue or organs</li> <li>4. Coma from overdose</li> <li>5. Death from overdose</li> </ol>

## Risk Management Activities

Recall that risk management is comprised of four individual activities: risk analysis, risk evaluation, risk control, and overall residual risk evaluation. Let us examine each of these activities and see how they might apply to a small example.

### Risk Analysis

The 14971 standard defines risk analysis as the “systematic use of available information to identify hazards and to estimate the risk.” In the infusion pump example above a single hazard was identified.

Risk analysis is concerned with the risk of harm. Hazards represent scenarios that could result in harm. The 14971 standard specifies that risk analysis is comprised of three steps:

1. Intended use and identification of safety characteristics;
2. Identification of hazards;
3. Estimation of risks.

#### Specifying Intended Use and Identifying Safety Characteristics

The intended use of the device is important to describe so that it can be distinguished from potentially harmful misuse of the device. Similarly, the safety characteristics define those limits within which a valid intended use of the device is determined to be safe. This step exists simply to define the limits of safe operation to facilitate identification of hazards that would take the use of the device outside those bounds of safety.

#### Identification of Hazards

Our infusion pump example only considered part of one hazard. (We will come back to why this might have been only part of one hazard.) The “identification of hazards” involves systematic and iterative updates throughout the life cycle of the product. In early phases, the hazard identification is all predictive. There is no prod-

uct and there are no designs or requirements. One can only predict what the hazards might be. That is not as hard as it sounds. Usually companies have a rich history with the product line or similar products and can predict hazards based on experience.

The early phase hazard identification, since it is largely predictive, should summon the experience and creative talents of not only engineers, but clinical specialists, regulatory and legal specialists (who may know of recalls or lawsuits related to product failures of similar devices), and marketing representatives (who may know of problems with legacy and competitive devices). One never knows where this kind of predictive knowledge will come from, so it is wise to cast a wide net!

In later phases of the development life cycle, hazard identification is still predictive, but it is predictive based on more detailed information on what the product is becoming, and how it was designed and implemented. The same mix of talents should be involved in these hazard identification sessions, but the contributions will become more technical and engineering centered.

One final word on the identification of hazards is warranted. As you will soon see, this process is rarely as simple as the table above initially would lead you to think. The analysis often leads to complex hierarchical relationships, in which it often is difficult to determine whether an event is part of a sequence of events, a hazardous situation, or whether it may be a separate hazard altogether. This is normal. One school of thought is that the semantics is really not important. The relative relationships and consistency of treatment are far more important than labeling an event with the “correct” term, if one even exists.

### Estimation of Risk

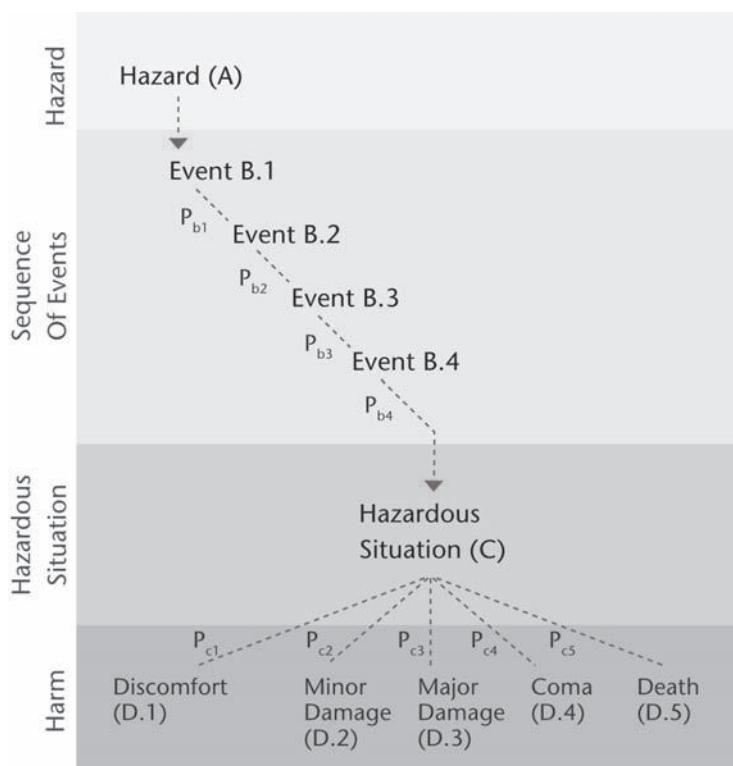
The hazard we identified in our example actually introduced five different harms. To consider the risk of these harms, we will need to consider the severity of the harms and their probabilities of occurrence.

### Quantitative Probability Analysis

For the time being, let us ignore the difficulty to be encountered in trying to assign probabilities to software errors, other systematic defects, or the user errors that initiated the sequence of events for that matter. The distribution of probabilities for the hazard analysis of Table 8.1 is shown in Figure 8.5.

Let us also assume for the time being that the team responsible for this hazard analysis is able to agree to a severity scaling for each of the harms in this example. Usually it's easy to get consensus on severity, but we will come back to that later. This figure shows that there are a number of probabilities involved in this analysis, more so than are implied by the simplified discussion in the standard.

Following the example, there is only a probability  $P_{B1}$  that the unexpected user programming would lead to a corrupted Limit\_Volume value (from Table 8.1). If the value is corrupted, there is a probability  $P_{B2}$  that the corrupted value would lead to a condition in which the pump would not shut off. That probability is not equal to 1 because the corrupted value could be too low, and the pump could shut off prematurely (suggesting another hazard).



**Figure 8.5** Distribution of probabilities in hazard analysis.

If the pump fails to shut off because of the corrupted value, there is a probability  $P_{B3}$  that the pump would run beyond its programmed limit. Note that probability isn't necessarily = 1. It could be that the majority of these therapies require 1 mL of infusions from a 1-mL syringe, or maybe there are other limitations in the system that could prevent a runaway pump from overinfusing. Note that this kind of analysis is factoring in the intended use of the device. Without an understanding of the intended use, this kind of analysis will be incomplete, and could be misleading.

Finally, if the pump runs beyond its programmed limit, the probability that it results in a patient overdelivery is  $P_{B4}$ . That probability is likely to be almost equal to 1, but not necessarily equal to 1. Again, in a situation in which almost all of the deliveries are equal to the volume of the syringe, there is no opportunity to overdeliver to the patient even if the pump attempts to run beyond its programmed limit.

That leaves us with an overall probability of reaching the hazardous situation of a patient being overinfused of  $P_{HS}$  which is:

$$P_{HS} = P_{B1} \times P_{B2} \times P_{B3} \times P_{B4}$$

This hazardous situation could result in one of the five identifiable harms in the analysis. The probability of each of those harms is equal to the probability of the hazardous situation, resulting in that particular harm multiplied by the probability

of that hazardous situation occurring. For example, from the figure the probability of death is given by:

$$P_{\text{death}} = P_{C5} \times P_{HS} = P_{C5} \times (P_{B1} \times P_{B2} \times P_{B3} \times P_{B4})$$

Since all probabilities are less than or equal to 1, these chains of multiplied probabilities can become very small if the individual probabilities are all significantly less than one. That leads us to an important observation: Hazardous situations that result from long sequences of events are much less likely to occur than short sequences of events—if the probability of transitioning from event to event is less than 1.

The probability that no harm results from the hazardous situation in our example is equal to:

$$P_{\text{no\_harm}} = 1 - (P_{\text{death}} + P_{\text{coma}} + P_{\text{major}} + P_{\text{minor}} + P_{\text{discomfort}})$$

This is a good cross-check or sanity check if quantitative probability estimation is possible (it is not always). If the probability of no harm is extremely high, one should consider qualitatively whether that seems reasonable. If “no harm” seems too likely from the analysis, probability estimates should be reexamined for why the probability of no harm is so high. It is also possible the  $P_{\text{no\_harm}}$  could work out to be a negative number. This could occur if the probability estimates are too conservative. An error that is frequently observed is that the individual harms are not considered to be mutually exclusive for the sake of analysis. For example, one might argue that if the resulting harm is death, then the patient may have been in a coma, which resulted from major organ damage. So, the logic goes, that for every occurrence of death, all of the other harms may have occurred, so the probabilities are cumulative, and so on. With that logic, every harm would have had some level of discomfort—and our probability of discomfort could end up at a value much higher than 1! When harms are a continuum as we have used in this example, it is the maximum, or final harm that occurs that should be considered in the analysis.

After this analysis, we are left with probabilities for each of the possible harms that we can anticipate resulting from our foreseeable “pump won’t stop” hazard. Before moving on to consider severity, let us zoom out a bit to see where our little risk analysis so far fits into the grand scheme of things.

In Figure 8.6 the hazard “pump won’t start” is examined a bit more fully. The preliminary analysis of Table 8.1 is represented by the leftmost series of branches in this hierarchy. Without going through the details of this figure, one can appreciate that the analysis of probabilities associated with a more realistic scenario like this are even more complicated than the analysis of probabilities for our preliminary analysis. For each branch of each of the sequence of events, a separate probability will exist for each of the hazardous situations shown. Further, each identifiable sequence of events that results in a hazardous situation will have its own mix of probabilities of that hazardous situation, resulting in each of the possible harms. For example, compare the probability of death for any of the first five sequences of events shown in this figure. Each of these sequences of events results in the pump running with no way to stop it. In the sixth event (pump overshoots limit before test) the pump does not stop when it should but it does not run endlessly out of

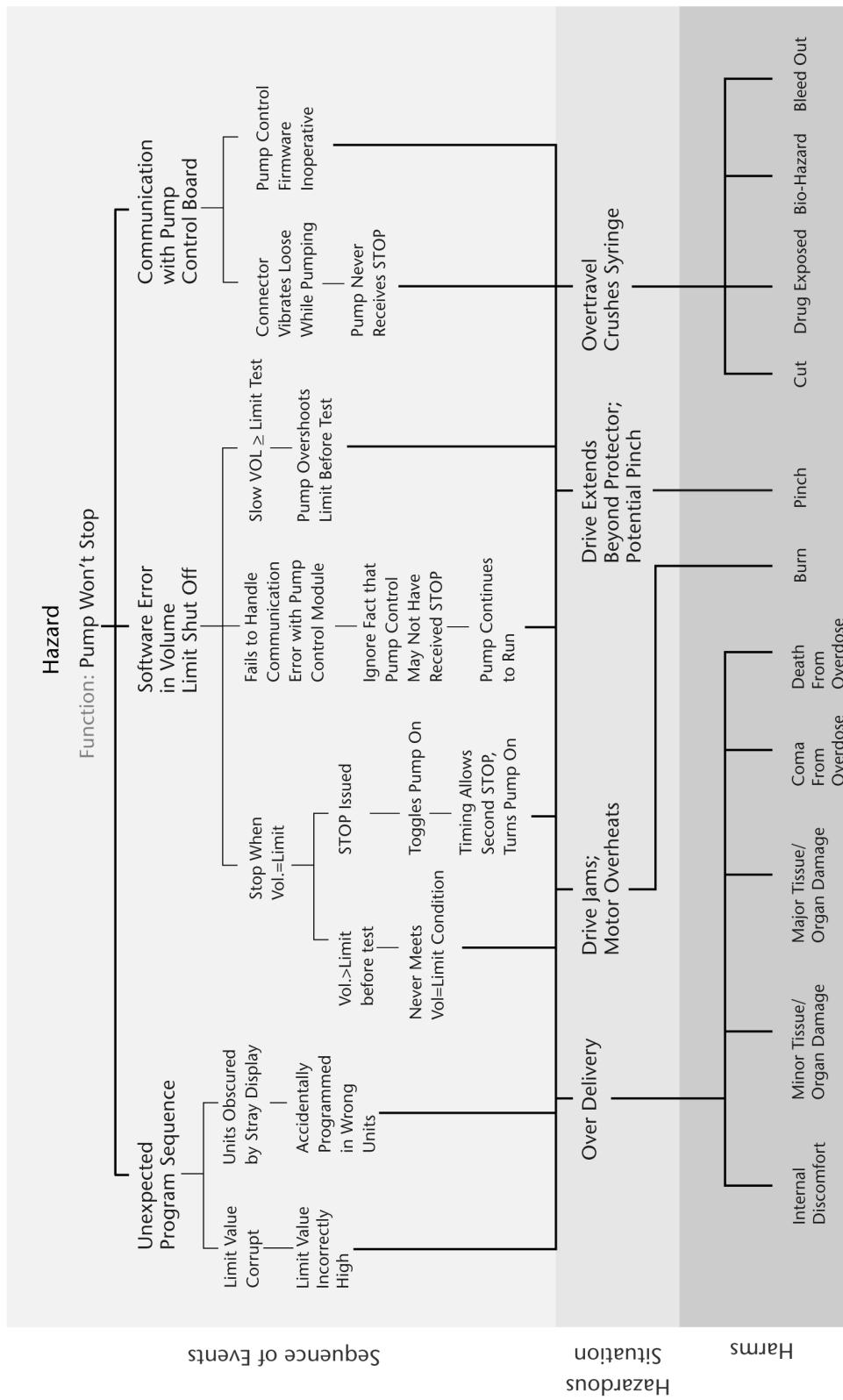


Figure 8.6 A more detailed hazard–harm analysis.

control. In this case the probability of an overdelivery in the microliter range is probable, but that small an overdelivery that results in patient death is very remote, and is much different from the probability of death for the other five sequences of events.<sup>1</sup>

One should not get too discouraged about the level of complexity yet. In the real world, most of these probabilities are not known quantitatively. The analysis of probabilities is not usually possible to do with real, fact-based probabilities. Many do attempt a quasi-quantitative analysis with estimated (often order of magnitude) ranges of probabilities. Almost as many go back to adjust those estimates when the end results of the analysis does not feel right. There is nothing wrong with cross-checks in calculations and comparisons of the results to what our “gut” tells us, but if the instinctive estimates always override the analytical results, one might question the process. In these situations in which the probabilities are always adjusted to yield an answer that meets the instinctive expectation, one might argue that the analysis of probabilities was a waste of time. It might well be a waste of time if we do not have some basis for the quantitative estimates of probability.

Since it is unusual to have good quantitative probability estimates to use in risk analysis (especially where software is involved), the concept of qualitative analysis of likelihood is attractive. In fact, the 14971 standard goes even, further stating that:

In the absence of any data on the probability of occurrence of harm, it is not possible to reach any risk estimate, and it is usually necessary to evaluate the risk on the basis of the nature of the harm alone.

That thinking is echoed in the 62304 Standard for Life Cycle Process, which states:

If the HAZARD could arise from failure of the SOFTWARE SYSTEM to behave as specified, the probability of such failure shall be assumed to be 100 percent.

Why do all that analysis of probabilities, when we do not even have probabilities available to use? Our time has not been wasted in this discussion because our thought process for qualitative analysis will be very similar, and is based on, the quantitative analysis above. The above breakdown of hazards, sequences of events, hazardous situations, and harms is a valuable skill to develop and a valuable piece of documentation to create to understand all the risks and their relationships. The value will become clear in the discussion of risk evaluation and risk control.

### **Qualitative Probability Analysis**

If you didn’t know it before, by now you should have figured out that the software engineers are not going to tell you that the probability of a software defect in the XYZ module is  $4.5872 \times 10^{-5}$ . (In fact the probability that they would give you that definitive an estimate is only infinitesimally greater than zero!)

14971’s inference that it may be “necessary to evaluate the risk on the basis of the nature of the harm alone” is attractive for software-based devices. This

1. An alternative way of analyzing this hazard would make the hazard description narrower in scope to distinguish between a pump that will not stop ever from a pump that simply overshoots the target volume but does stop eventually. That is, “Pump Never Stops” and “Pump Overshoots Delivery Volume Target.”

approach solves a number of issues in risk analysis and risk control related to probabilities, but still leaves a few residual problems.

### Ignoring Probability

One can ignore probability by setting all probabilities to a value of 1. What remains is a hazard analysis similar to the one in Figure 8.6 in which no probabilities are shown. There are 11 potential harms shown in that figure. Assume for now that the risk management plan defines five severity levels. The 11 harms get sorted into those five severity levels. Now assume that the risk acceptance definition in the risk management plan states that the device cannot be released without some acceptable level of control on the top four severity levels.

What does risk control look like if we ignore probability? In this type of analysis, risk controls can only work to reduce the potential severity of the harm resulting from a failure. Going back to our syringe pump example, the hazardous situation of “overdelivery” can result in five harms of different severity level. Suppose we add a risk control measure to limit how long the pump can run beyond its prescribed volume limit. Depending on how well we do that, it would appear that we may have shifted the severity of the harm of the hazardous situation downward—a good thing. In fact, that is a key value of ignoring probability: we focus on controlling the severity of the resulting harm.

So what is the problem? I thought not. There is still some probability that it could still happen (maybe our control measure fails, or a very concentrated drug is being infused). If a patient death could still happen, then we haven’t removed death as a harm resulting from this hazardous situation. What we really have done is shifted the *probability* away from death to some harm(s) of lower severity. Since there is still some probability of an overdelivery resulting in death, and this type of analysis ignores probability, it is almost impossible to control risk of harm unless we actually do—somehow—consider probability.

### Qualitative Probabilities

Purely qualitative probability analysis does not use any numeric values for probability of harm. Probability estimates are simply relative estimates (i.e., one event is more probable than another, but no values are associated with how much more probable the event is, or what the absolute probability is). The 19471 standard gives examples of high, medium, or low, or alternatively, frequent, probable, occasional, remote, or improbable as examples of qualitative probabilities. The names and number of qualitative probability levels is not too important within reason. Too many levels will make it difficult to distinguish between levels, and too few levels will not give enough resolution to adequately estimate the effect of the risk controls. What is very important is to consistently apply the probability levels for similar situations. This implies that the rules should be well documented (in the Risk Management Plan).

Once discrete qualitative probability and severity levels are assigned, risk evaluation tables can be constructed as shown in Table 8.2. Each cell of the table represents a possible risk level. Those cells that are shaded in the table represent unacceptable risk levels (as defined in the risk management plan). The risks of the

individual harms (represented arbitrarily by R1 through R6 in the table) are compared to their placement in this table to determine whether something must be done to control the risk down to an acceptable level.

Seems simple again, right? The overall process from the 14971 standard is a great suggestion, but the realities of applying this process are considerably more complicated in real applications. Consider again Figure 8.6. Eight separate sequences of events in the figure potentially lead to an overdelivery hazardous situation. Each sequence of events has potentially a different probability of resulting in an overdelivery. There are five separate harms that could result from the overdelivery in the harms band of the figure. The probability of any given harm is equal to the probability of occurrence of the hazardous situation times the probability of the hazardous situation resulting in that particular harm:

$$P_{\text{harm}} = P_{\text{hazardous\_situation}} \times P_{\text{harm\_from\_hazardous\_situation}}$$

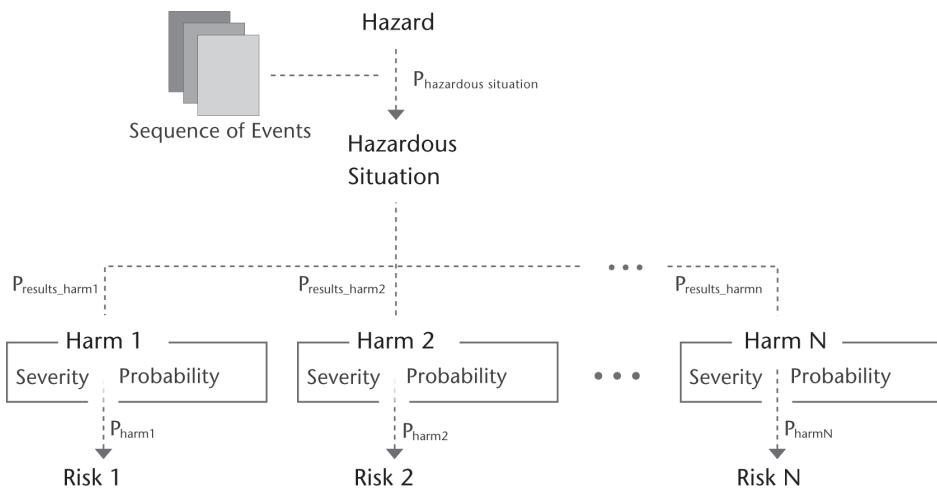
Let us assume three discrete levels of probability of low, medium, and high (L, M, and H). There are then  $3 \times 3 = 9$  possible probability combinations (three for the  $P_{\text{hazardous\_situation}}$  and three for the  $P_{\text{harm\_from\_hazardous\_situation}}$ .) That gives six unique possible probabilities of harm, assuming that the equivalent qualitative values (i.e., L, M, and H) are equal (i.e., an L probability for  $P_{\text{hazardous\_situation}}$  is qualitatively equal to an L probability for  $P_{\text{harm\_from\_hazardous\_situation}}$ ). The possible probabilities that make up the probability of harm:

1. LxL;
2. LxM MxL;
3. LxH HxL;
4. MxM;
5. MxH HxM;
6. HxH.

Now consider, if five levels of probability are chosen in a qualitative system. If one *separately* considers the probability of a hazardous situation occurring, and the probability of that situation resulting in a given harm (as recommended by the 14971 standard), then  $5 \times 5 = 25$  different probability answers are possible, with 15 unique values.

Substituting qualitative probability values succeeds in providing understandable *relative* probability values, and the analysis starts to make sense. The overall risk evaluation (or residual risk evaluation), however, will still be a challenge. Refer back to Figure 8.6 once more. To evaluate the overall risk of “death from overdose” (which clearly is important to do) involves analyzing the eight individual sequences of events that lead to the hazardous situation of “overdelivery” and the probabilities for each of those eight sequences of events leading to the harm of “death from overdose.” It is beyond the scope of this text to treat that analysis mathematically, or more correctly, pseudomathematically.

The 14971 standard implies (i.e., it implies it by not offering an alternative) that risk analysis should be dealt with at the level of individual combination of hazard, sequence of events, and hazardous situation level as shown in Figure 8.7.



**Figure 8.7** Hazard analysis approach to risk analysis.

This is very similar to Figure 8.4, except that it acknowledges that a given hazardous situation could result in a number of harms (or the same harm at differing severity levels), each with its own probability. This, at least is easy to understand. It ignores the overall evaluation which will be dealt with shortly). However, if a hazard can result in a significant, unacceptable risk on its own, it will certainly stand out in this type of analysis.

This is a good representation of the analysis, but are we expected to create diagrams for each combination of hazard, sequence of events and hazardous situation? Probably not. Practically speaking, the analysis is probably better handled in a table format.

*Qualitative probability of software failure.* If, based on the above discussion of qualitative probabilities, one decides to implement qualitative probability scaling for software failures, the question still remains how do we determine what is a low, medium, or high probability of failure? It has been stated a number of times that nobody really knows the actual probability of a software failure. However, we do know a large number of factors that might increase or decrease the probability. Let us examine what some of these factors are, then think about how to leverage this information into a methodology that can be used to consistently assign qualitative probability levels for similar situations.

The following factors are likely to increase the probability that a given software item or function will fail:

- Size of the software item;
- Complexity of the software item;
- Inexperience level of developer;
- Low skill level of developer;
- Changes made late in life cycle;
- Changes made in stressful (usually schedule) situations;
- Functionality not well understood/documented;

- Functionality not well exercised in trial-run, or ad hoc, testing;
- Software item has history of problems;
- Regression history shows item has not been tested recently;
- Item not tested;
- Inexperience level of the tester;
- Skill level of the tester;
- User confusion possible.

This is not an exhaustive list, but it is adequate to make the point that we want to make here. We know that each of these factors increases the likelihood that a specific item of software will fail. We also know intuitively, that when more of these factors are present, the probability is higher than when there are fewer factors present.

One approach to assigning qualitative probability levels is to define a set of rules for that assignment based on factual qualitative information. The factual information might be an assessment of whether the item under consideration is subject to any of the factors in your list of factors that affect probability. (By the way, do not use the above list as a totally comprehensive list. Use it as a starting point for your own list of factors.)

The rules simply set thresholds of how many probability factors would put an item in a given risk category. For instance, one might say that zero or one factor is low probability, two or three factors are a medium probability, and four or more factors is high probability. The granularity of the factor list and one's experience in using it will determine where these thresholds should be set.

Alternatively, one might consider prioritizing or weighting the factor list since it is likely that not all factors contribute equally to the probability of failure. More complex rule sets would be necessary for determining a final qualitative probability based on a more complex factor list.

What is very nice about this approach is that the probability analysis actually suggests potential control measures for reducing the probability. Consider a software item that is implicated in a sequence of events leading to a harm. The probability of failure from a qualitative analysis of that software item makes the risk of the harm unacceptable. Suppose the factors leading to the high probability estimate were the complexity of the item, the inexperience level of the developer, and the lack of testing. Reasonable control measures would simply be to reengineer the item to reduce its complexity (if possible), subject to the item for review to a more experienced developer, and increase the amount of testing of the item, presumably by an experienced, skilled test team.

### Severity

There usually is much less debate over severity. Whenever there is a debate, it usually because the harm is too broadly defined. In our example, if we just had “tissue damage” as a resulting harm, it might have spawned debate over how severe the damage might be. One can simply split that particular harm into multiple levels of severity as we've done in this example to settle the debate.

It is also possible to have too many harms at too many severity levels. The number of levels can have an effect on the risk management process. There is a tendency to define many levels for very specific situations. However, the problem with too many severity levels is that unless something different will be done in the upcoming risk estimation, evaluation and control for the different levels, then an overabundance of severity levels does not add any value to the process.

How do you know if you have too many, or too few levels of severity, or if you have split your harms into the right number of categories? There isn't a right and wrong answer to this question, but some answers are "more right" than others. There are several things to consider when deciding how many levels of severity should be defined in the severity scale.

1. The reason for assigning a severity is to assess the risk, determine if it is acceptable, and if not, to control it to a level that it is acceptable. So, one needs enough levels in the severity scale to have some acceptable, and some not acceptable (i.e., a minimum of two).
2. Risk management for software-driven systems relies heavily on managing severity downward. In the best case, we can manage severity down to an acceptable level, making probability a moot point. However, it is more common that control measures can be added that reduce severity somewhat, but not to the point of acceptability on its own. In those cases, other control measures may be able to reduce probability to an acceptable level of risk. Enough levels of severity are needed for the risk evaluation to determine if the risk is near the threshold of acceptability or not.
3. Perhaps most commonly, four to six levels are used, but it is not unheard of for some analyses to use 10 or more levels of severity. If it becomes difficult to determine what level of severity to use, or if the overwhelming majority of harms cluster in a much smaller number of severity levels, there are probably too many levels of severity for practical analysis.

One might choose to embed an indication of severity in the name of the harm (e.g., "minor burn," or "serious shock"). Alternatively, one might just choose a harm of "burn" and "shock" and assign a severity to them in the risk analysis. The example of Figure 8.6 uses a combination of methods. The harms related to the hazardous situation of "overdelivery" have implied severities in the names (discomfort, minor tissue damage, major tissue damage, coma, and death). Other harms such as "burn," "pinch," and others are not specific in their severity. Not all sequences of events that result in a given harm result in the same severity of that harm. In that respect, Figure 8.6 is overly simplified showing a single mapping of a hazardous situation to a single harm. In reality, that heavy line represents that different sequences of events could lead to that harm, but at different levels of severity.

The objective of risk analysis is not really to come up with the best, most complete risk/harm scoring system, or to divide it into more levels than anyone else. The objective is to choose a number of levels that easily supports the risk evaluation, and does not get in the way of the risk analysis. If it feels like time is being wasted splitting hairs over what severity level to use, then time probably is being wasted and the number of levels should be changed.

The 14971 standard offers two examples of scaling levels in Annex D. One example is simply:

- *Significant*: Death or loss of function or structure;
- *Moderate*: Reversible or minor injury;
- *Negligible*: Will not cause injury or will injure slightly.

Another example offered by 14971 is:

- *Catastrophic*: Results in patient deaths;
- *Critical*: Results in permanent impairment or life-threatening injury;
- *Serious*: Results in injury or impairment requiring professional medical intervention;
- *Minor*: Results in temporary injury or impairment not requiring professional medical intervention;
- *Negligible*: Inconvenience or temporary discomfort.

It is also common to use numeric values to represent the severity levels. The names or numbers are not important as long as their relative position on the severity scale is easy to remember. As with probability, the number and naming of the severity levels is not too important within limits. Consistency in the application of the severity levels is the most important factor, and the rules for determining the severity level should be well documented in the risk management plan.

### Detectability

A practice that is sometimes found in the estimation of risk is to factor in the detectability of the hazard, hazardous situation, or even the sequence of events leading to the hazardous situation (let us call these three collectively a “failure”). The theory is that if the failure is detectable, it is somehow of lower risk. This practice (in my opinion) came from process validation for production processes and the failure modes and effects analysis (FMEA) used to identify hazards and estimate their risks. In a manufacturing process which is being closely monitored by production personnel, and whose outputs are often checked by quality control personnel, a detectable failure is lower risk, because people are paid to watch for failures and there is a reasonable assumption that they will do something about them.

In the case of a medical device, the assumption by the caretaker or patient usually is that the device will work, and rarely are they prepared to watch for device failure, nor are they trained to do the right thing in a way that would reduce the likelihood or severity of harm resulting from a failure.

Consider a truck whose breaks have failed that is rolling across a parking lot toward a cliff. To everyone in the parking lot, the sequence of events and likely hazardous situation about to unfold is very obvious (i.e., detectable). That blatant detectability is not likely to do much to reduce the probability of the events leading to the hazardous situation of the truck falling off the cliff.

Now consider the same situation, and the first event in the sequence of events is that a leak develops in the hydraulic brake line. There are two detectable items, a

puddle of brake fluid in the parking lot, and a sensor on the truck's dashboard indicating imminent brake failure. Both of these items make the failure detectable, but the dash indicator is more likely to get a response from the driver to reduce the probability of the hazardous event than the puddle in the parking lot is.

Detectable failures will not reduce the probability or severity of a risk unless someone does something about the detectable event. Does that mean detection has no role in risk management? No, detectable failures are always better than undetectable ones. However, the reason they are desirable is that they make corrective risk control measures possible which reduce the risk, not that detectability by itself reduces risk. In my opinion, blindly reducing a risk estimate because the failure is detectable is wrong and invites an incorrect risk evaluation. That is, the risk may be seen as acceptable because of detectability, thus allowing the risk management process to skip adding a risk control measure. Unfortunately, it is the risk control measure that would actually have reduced the risk.

## Risk Evaluation

Recall that one of the components of the risk management plan was the definition of acceptable risk. That definition will become extremely important for risk evaluation. Acceptable risk definitions may be similar to the example shown in Table 8.2 in which the shaded risk cells represented unacceptable risk. For larger risk matrices (i.e., more levels of probability and severity than used in our example), acceptable risk may be defined at more than the two levels of acceptable and unacceptable. In fact, Edition 1 of 14971 introduced a third level of acceptability referred to by the acronym of ALARP, which stands for As Low As Reasonably Practicable.

Risk acceptability does not have to be in the form of a table. When numeric values are given to severity levels and either quantitative or qualitative probability levels, then risk is often expressed as the arithmetic product of severity multiplied by the probability. In these cases, the risk acceptability may be as simple as a numeric threshold above which the risk is unacceptable.

The introduction of an ALARP zone of acceptability leads to an interesting question of how many zones, or levels, of acceptability should be defined. The answer to this varies depending on the risk tolerance profile of the device being analyzed and by the practicalities of dealing with acceptance zones. With two zones, it is very clear that if the risk is unacceptable, then controls must be introduced until it is acceptable. When a third ALARP zone of acceptability is introduced, the risk management plan must define the limits of the new zone, but equally (if not more) importantly the plan must define what is to be done with risks in the new zone. The same would be true if additional zones of acceptability are introduced. One might

**Table 8.2** Risk Evaluation Table for Qualitative Risk Assignments

<i>Probability</i>	<i>Severity</i>		
	<i>Negligible</i>	<i>Moderate</i>	<i>Significant</i>
High	R1		R3
Medium		R2, R6	
Low	R5	R4	

surmise that for risks in the ALARP region the severity and/or probability should be controlled downward until the residual risk is acceptable. If that is not possible within reasonably practicable controls, then the risk is defined as acceptable even though the risk may not technically be in an acceptable zone.

The result of the risk evaluation activity is an identification of those risks that need additional controls, documentation of those risks deemed to be acceptable, and the rationale that lead to that conclusion of acceptability. Unacceptable risks are subjected to the risk control activity, then resubmitted to this risk evaluation activity to determine whether the residual risk post-control is acceptable. If not, the cycle repeats until the risk is acceptable.

Earlier, some criticism was leveled at risk management processes that result in elaborate arithmetic estimates of risk. When those estimates indicate that many controls are needed, the estimates are reestimated to manage the *workload* down to an acceptable level. That is not the point of risk management.

At the risk (no pun intended) of sounding hypocritical, there sometimes is an advantage to reestimating risk, adjusting the probability/severity definitions, or redefining risk acceptability in the risk management plan. Those situations are the opposite of re-estimating to reduce workload. In these situations a risk estimate falls into a region of acceptability, but the result does not match our “gut instinct.” In other words, it feels like something should be done to reduce the risk, even though the analysis indicates that the risk is acceptable. In situations like this, why wouldn’t you do whatever it takes to reduce the risk? It may take a revision of the risk management plan, and some reassessment of risks already analyzed. But then, who would want to explain that a patient injury occurred simply because its risk fell in an unshaded box (acceptable risk), and nothing was done to control it—even though it seemed like something should be done.

The bottom line is that we should let common sense prevail in risk evaluation. There is a great deal of imprecision in the risk management process when we deal with estimated quantitative or qualitative probabilities. There is little wisdom in adhering to a rigid plan that documents an intent *not* to address significant risks.

## Risk Control

Let us review the process that leads up to risk control.

- Hazards have been identified.
- Risks associated with harm(s) resulting from each hazard have been estimated.
- Risks have been evaluated. Some of them were found to be unacceptable according to the definition of acceptability in our risk management plan.

So, now it is time to design (or identify preexisting) controls for those unacceptable risks. The risk control activity is defined by 14971 as the activity “in which decisions are made and measures implemented in which risks are reduced to, or maintained within specified levels.” We will refer to these measures as risk control measures or RCMs. In some FDA guidance documents they are referred to as protec-

tive measures. Since risk is a combination of severity and probability of harm, RCMs can focus on controlling either severity or probability to reduce the risk.

Not all RCMs are created equal; some are better than others. For our purposes in this text, let us consider five different types of RCMs presented in their order of effectiveness in controlling risk:

1. Inherently safe design measures make the risk impossible or nearly so. For example, suppose a blood warmer, with a functional hazard of “heater stuck ON,” resulted in a hazardous situation of “blood cells damaged,” which ultimately leads to several harms of varying severity. If the heating element of the warmer is physically incapable of heating higher than 98.6°F (by any means), then the design is inherently safe. No matter what sequence of events takes place that would lead to “blood cells damaged,” the probability will be 0 or almost 0.
2. Preventive measures reduce risk by eliminating or reducing the probability that a sequence of events leads to a hazardous situation, or the probability that the hazardous situation results in a specific harm. A cover on an emergency stop button is a preventive measure. It helps *prevent* the hazard of a prematurely terminated therapy caused by a sequence of events initiated by the emergency stop button being pressed accidentally.

Preventive measures in software are common. Cross-checks on critical calculations or bounds checks on the results of calculations to check for potentially dangerous incorrect results are preventive measures. Checksums on the program code prevent running with corrupted code. Checksums or cyclic redundancy checks (CRCs) on communications prevent an inappropriate device response due to a corrupted communication similarly are preventive.

3. Corrective measures detect and take corrective action if a hazardous situation is about to occur. Some watchdog timers (WDTs) are good examples of corrective measures. WDTs can be triggered for a variety of reasons (the processor hasn’t reset the WDT in less than the threshold time for the WDT to trigger, or a communication hasn’t been received from another processor in the system, etc.) When a WDT has triggered, the sequence of events leading to the hazardous situation has already started. The WDT triggers, forcing the processor to reinitialize, and, hopefully, resolve the issue and continue operating.
4. Mitigating measures (or mitigations) reduce the severity of harm if a hazardous situation occurs. A WDT that detects a timing issue as described above and halts the processor to prevent an overdelivery of drug or energy to the patient has not corrected the problem as a corrective measure would. Instead, it simply stops everything to limit the severity of harm that could result from the hazardous situation of a runaway processor.

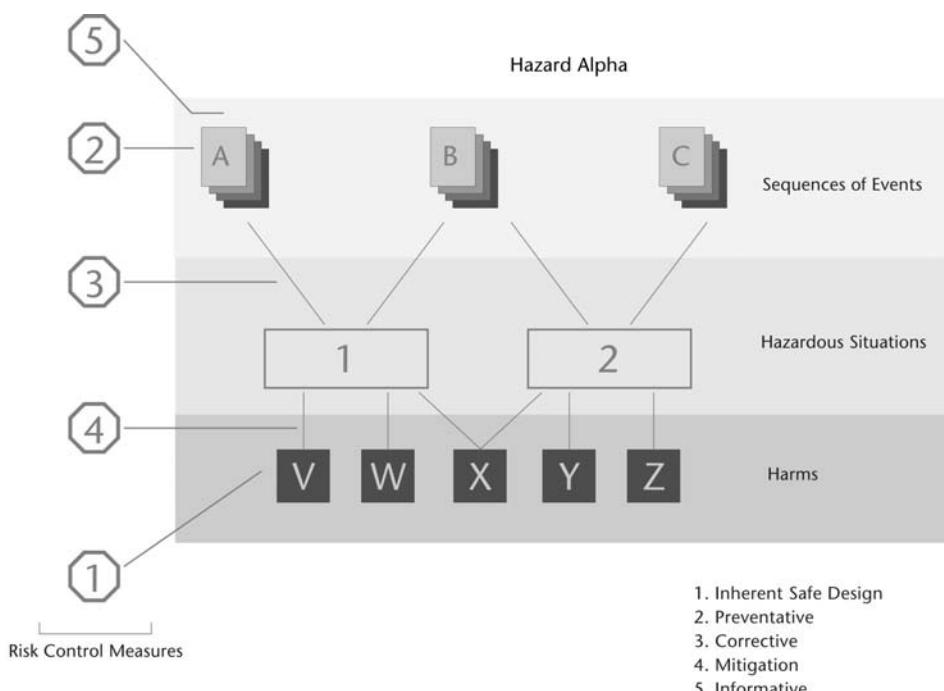
In the hardware world, fuses (depending on their use) are good examples of hazard mitigations. The hazardous situation of a shock has already occurred, the current limit is reached and the fuse blows to reduce the severity of the resulting harm. Note, however, that if a fuse simply blew because it detected too much current to a motor winding, then it may be

preventing hazardous situations related to overheated motors, wiring and driver boards.

5. Informational control measures are the control measures of last resort when something substantive cannot be done to reduce risk by other means. Informational control measures include product labeling, warnings in user manuals, user prompts and messages, limitations in intended use, and user training. Admittedly, sometimes this is all that can be done to reduce a risk, but overdependence on informational control measures is not a good indicator of a safe device. Informational control measures may never be conveyed to the user (lost user manuals), forgotten (training), or not understood (labeling in a foreign language or obtuse wording).

Why is it important to understand these different types of risk control measures? It is doubtful that anyone will ever ask you to classify risk control measures by type. It is likely that if you did try to classify RCMs, you would run into differences of opinion about what type each RCM is. Without adding more confusion by getting into the fine points about the differences, let us just accept that there is room for debate. What is important is to recognize that there are differences, and some RCMs are better than others.

Figure 8.8 shows how the various types of RCMs relate to the progression of hazards from the initiating sequence of events to harms. It is interesting to note that both informative and preventive measures are at the front end of the chain acting to prevent the initiating sequence of events from starting. It could be argued that infor-



**Figure 8.8** How RCMs relate to hazards, hazardous situations, and harms.

mative control measures are preventive. They are just particularly weak preventive or defensive measures.

The difference between preventive measures and corrective measures has more to do with what end of the sequence of events chain they act to control. The mitigations are at the tail end of the chain. They do not come into effect until the hazardous situation has already occurred and harm is underway. They act to shift the severity of the harm resulting from a hazardous situation from higher severity harms to lower severity harms.

Inherently safe design is shown here as directly impacting the harm. In other words, if the design is inherently safe, the probabilities of reaching a hazardous situation are of minor interest because the resulting harm cannot occur because of inherent safety.

In Table 8.3, the type of control offered by a risk control measure is considered. For the purposes of this table, probability means the combined probability of the sequence of events resulting in a hazardous situation and the probability of the hazardous situation resulting in a particular harm. As one can see, inherently safe designs act to control both probability and severity. Preventive and informative controls both act to reduce probability only. A mitigation controls the severity of a harm resulting from a hazardous situation. Corrective actions could reduce either the probability of a risk, the severity of a risk, or a combination of the two.

Why does this matter? It matters because risk is usually better controlled by reducing the severity of a resulting harm than the probability of the harm. Of course there are exceptions, but especially with software controlled devices in which the probabilities are so nebulous, one has a much clearer understanding of the amount of control exerted on a risk if severities are manipulated rather than probabilities.

Risk control consists of an analysis of the relationships between the sequences of events, hazardous situations, and harms, then identifying preexisting controls, or creating requirements for new controls that adequately manage the risk to an acceptable level. Oftentimes, one will find that a number of risk control measures have already been designed into a device. Sometimes this is due to simple common sense. Sometimes risk controls are designed into subsystems that are included in a device, and sometimes risk controls are carried over from similar devices in the company's product line.

The risk control activity is just as important as the analysis and evaluation. It needs to be taken seriously and given adequate thought so that no reasonable opportunity is overlooked for designing in control measures that will make the device safer. Too often, an inordinate percentage of risk control measures are listed as "labeling," "testing," and "watchdog timer." Granted, there are risks for which

**Table 8.3** How Different Types of RCMs Control Risk

RCM Type	Reduces Probability of Risk	Reduces Severity of Risk
Inherently safe design	Yes	Yes
Preventive	Yes	No
Corrective	Possibly	Possibly
Mitigation	No	Yes
Informative	Yes	No

there are no good control measures other than these three or that these three measures are in fact the best control measures given the risk. On the other hand, in too many analyses, these three represent easy answers that are filled into a chart simply to fill in the chart.

In a project we participated in a number of years ago, we noticed that our own risk management documentation had a large number of risks that were only controlled by a watchdog timer. We carefully reexamined each of the risks and came to the conclusion that there really was very little more we could do other than depend on a watchdog timer to save us from resulting harm. However, as we repeated that analysis we asked ourselves, “What exactly are we expecting the WDT to detect and how exactly are we expecting it to respond?” In so doing, we identified roughly 15 individual requirements for our WDT design that were necessary for the watchdog to, in fact, protect us from the hazardous situations we had identified. We learned from this to always ask ourselves what and how we are expecting the risk control measure to reduce the risk. If this is not done, and the detailed requirements for the control measure are not recorded, it would be far too easy to think that the risks are controlled simply because a control measure by that name existed, even though it may not have had any effect whatsoever on the risk to be controlled.

Testing, as mentioned above, is often overused as a risk control measure. Is software testing a risk control measure for harms that result from a software failure? The answer is, yes, it is a risk control measure—it is a corrective control measure for the design process that creates the software. It acts to reduce the probability of the software failure, because presumably the testing will identify defects in the software that will be corrected before it is released for use.

However, since testing only acts on probability, it is a weak control measure for hazardous situations resulting from software failure. The previous discussion on the probability of software failure suggested that no one knows the probability of a specific software failure—maybe not even within an order of magnitude or two. Also up for grabs are the probabilities that our testing will detect a defect, and that our corrective actions will eliminate the defect. In other words, when testing is used as a risk control measure, we take a moderately random number estimate used for probability of failure, mix that with a few other random numbers relating to our test effectiveness, and come up with a new random number which is our residual risk.

Yes, there are certain types of risks that may only be controlled by testing. Further, the point of this discussion is not to discourage software testing. Clearly, more testing should result in fewer defects in the finished product. However, one should accept software testing as the flawed process that it is, so that one does not rely too much on its effectiveness to control critical risks. Software testing is just as subject to design flaws as the software development process is; in fact, it has been pointed out numerous times just how similar the development and test processes are. It is usually accurate to say that if a control measure exists within the device (rather than within a design process, such as testing) that it will be more effective at controlling the risk.

### Overall Residual Risk Evaluation

Assume that the risk management team has completed the risk management process for each hazard that could be predicted or identified. The risk controls were identi-

fied, and on a pathway-by-pathway basis working with diagrams like Figure 8.6, or in a tabular format. The individual ways a hazard could result in harm were analyzed and controlled. The residual risks were rationalized to be acceptable. Is the job done?

Unfortunately, there is one very difficult activity yet to complete: that of the overall risk evaluation. Overall risk needs to be considered at two more levels. First (level 1), it must be determined if, for a given harm, its *combined* risks for all possible occurrences for a given hazard are acceptable. Second (level 2), a given harm may result from multiple hazards in a device. Therefore, it must be determined if the combined risks of each harm from all possible hazards is acceptable.

What exactly is meant by this? Referring back to Figure 8.8, consider Harm “X.” The two hazardous situations 1 and 2 both could result in Harm X. Further, both hazardous situations have two possible sequences of events leading to them. In total, there are four possible ways that Hazard Alpha could result in Harm X.

So far in the discussion risk analysis evaluates and controls risk for each pathway independently. Overall residual risk evaluation (level 1) considers if the residual risk of Harm X given the probabilities of all four pathways is still acceptable. Level 2 overall residual risk evaluation extends that analysis to all pathways to a given harm for *all* hazards.

Still referring to Figure 8.8, consider a situation in which the pathway from sequence of events A leads to hazardous situation 1, and results in Harm X. The risk of Harm X from that pathway was initially thought to be unacceptable, and through several iterations, enough risk control measures were added to reduce the risk to a “just acceptable” level. Now that we look at the overall risk, we realize there are three other ways related to Hazard Alpha that we could result in Harm X. Is its residual risk still acceptable?

That is a good question. How does one approach that analysis? Let’s not even start the probability discussion this time. Annex D of 14971 does allow for the fact that the analysis is difficult and offers some guidance on how to evaluate the overall residual risk. For now, consider the situation in which the overall residual risk is not acceptable. One could presumably consider additional risk controls to further reduce the risk, but then, presumably, the risk had already been reduced to an ALARP level, right? If that is the case and risks have all been reduced as low as reasonably practicable (ALARP), then what? In that case, 14971 suggests:

If the overall residual risk is not judged acceptable using the criteria established in the risk management plan, the manufacturer may gather and review data and literature to determine if the medical benefits of the intended use outweigh the overall residual risk. If this evidence supports the conclusion that the medical benefits outweigh the overall residual risk, then the overall residual risk can be judged acceptable. Otherwise, the overall residual risk remains unacceptable. For an overall residual risk that is judged acceptable, the manufacturer shall decide which information is necessary to include in the accompanying documents in order to disclose the overall residual risk.

Assessing medical benefits compared to the overall residual risk is beyond the scope of the discussion here. Realize, however, that sometimes it is possible that an unacceptable risk is acceptable.

Let us return to the question of how the overall risk is evaluated. This seems like a daunting task for which we have few tools to assist us. In fact, in Annex D, 14971 says, “There is no preferred method for evaluating overall residual risk and the manufacturer is responsible for determining an appropriate method.”

The standard offers several ideas for approaching the analysis, but, for the most part the ideas presented just add more information to the analysis, and don’t really offer guidance on how to evaluate the residual risk. Two of the ideas are to use an event tree or fault tree to aid in the analysis. The approach described so far already incorporates the tree approach to understanding the relationships of the risks of harm. The real problem is the overall magnitude of the risk without any really good quantitative estimates of probability. Fault trees are a great tool, but without quantitative estimates of magnitude of risk, overall evaluation will be difficult.

The standard also suggested that a review by application experts may be useful in assessing the overall residual risk. This idea has some value; however, one should realize that it is simply shifting the responsibility of the overall assessment to someone else with a different set of skills without offering them any additional tools for the assessment. From 14971 Annex D:

#### D.7.8 Review by application experts

An assessment of the benefits to the patient associated with the use of the device can be required in order to demonstrate acceptability of the device. One approach could be to get a fresh view of the overall residual risk by using application specialists that were not directly involved in the development of the device. The application specialists would evaluate the acceptability of the overall residual risks considering aspects such as usability by using the device in a representative clinical environment. Then, evaluation of the device in the clinical environment could confirm the acceptability.

This approach seems to be backed by a lot of common sense, but one would probably not want to burden the clinical application specialists with every single assessment of overall residual risk.

Since the standard offers little guidance in this evaluation, let us consider a few off-standard methods that may be useful.

*Proper presentation.* This is not so much a method for overall residual risk evaluation as it is a technique to assist that evaluation. Attempting to perform an overall evaluation for a particular risk of harm that could be spread across dozens of pages of hazard analysis breakdowns is difficult simply because the information is so scattered. A more effective presentation would be to collect all of the potential risks related to a specific harm on a single page (or as many as it takes). Certainly, the overall evaluation will be more effective with the data collected in one place.

*Rule-based.* The details of this method will depend on what methodology is chosen for scaling risk and determining its acceptability. Consider a methodology in which three zones are defined for determining the acceptability of a risk: acceptable, unacceptable, and ALARP. A rule-based system for overall residual risk evaluation

would include rules for determining the acceptability of the overall risk. For example:

- *Rule 1:* If all the residual risks for a given harm are in the acceptable region, then the overall risk shall be considered to be acceptable.
- *Rule 2:* If any of the residual risks for a given harm are in the unacceptable region, then the overall risk shall be considered to be unacceptable.
- *Rule 3:* If all but one of the residual risks for a given harm are in the acceptable region, and the exception is in the ALARP region, then the overall risk shall be considered to be ALARP.
- *Rule 4:* If two or more of the residual risks for a given harm are in the ALARP region, and all other residual risks are acceptable, then the assessment of overall residual risk shall be referred to application specialists for their review of acceptability.

This approach is simple, easy to understand, easy to explain, and easy to document. Is it too easy? The method is actually rooted in some quasi-quantitative logic which goes like this: Assume that the probability of risk in the three zones of acceptability can be represented by ranges of probabilities (even though we do not really know what the actual probability is). Let us say for the sake of argument, that the average probability for acceptable, ALARP, and unacceptable risks are  $10^{-8}$ ,  $10^{-5}$ , and  $10^{-2}$ , respectively. Further, let us assume that the individual pathways leading to the harm under consideration are independent. (This is probably not actually true, but this assumption probably will not skew the results of the following simplification that this will allow.) Then the total overall residual risk can be approximated by the sum of the average probabilities for each individual risk of the harm being considered.

Given this logic, one can see how the rules make some sense. Consider Rule 1. If all of the independent residual risks are acceptable, and their probabilities can all be approximated by the average probability of the acceptable region ( $10^{-8}$ ), then it would take  $10^3$  (i.e., 1,000) individual acceptable risks to sum up to the average probability of the ALARP region. That is probably a safe assumption. A similar analysis for Rules 2 and 3 will reveal that the overall residual risk will be dominated by the one unacceptable or ALARP risk, and thus can be approximated by the risk associated with the dominant independent risk. Rule 4 breaks with this logic a little. Consider a situation in which two ALARP risks exist in the overall analysis for a given harm. Using our average estimates of probability of  $10^{-5}$  for the ALARP region, the total probability would still only be  $2 \times 10^{-5}$ , which is far from the average values of acceptable or unacceptable risk probabilities. So, logically and arithmetically, one could make an argument that more than one ALARP risk for a particular harm would result in an overall risk of ALARP. In this case, the conservative nature of the person who wrote Rule 4 requires additional analysis before allowing more than one ALARP risk to be considered ALARP overall.

### Relative Comparisons

This approach is somewhat similar to the rule-based approach, and the logic behind it is based on the same quasi-quantitative analysis.

For a given harm for which the overall residual risk is being evaluated, the individual risks are collected and ordered from highest risk lowest risk. Individual risks are grouped with other risks of approximately the same probability. This approximate probability may be the acceptability region that the risk falls into. As in the rule-based method, one can safely assume that the overall probability, and therefore the overall residual risk (since the severities are equal) are dominated by the highest risk group of individual risks. That reduces the evaluation to determining whether the combined risk of the highest risk group is acceptable overall.

### Safety Assurance Cases

One method for assessing overall risk is the development of a safety assurance case which is sometimes referred to as a safety case. IEC/TR 80002 describes safety cases in its Annex E as “a structured argument supported by a body of evidence that provides a compelling, comprehensible and valid case that a MEDICAL DEVICE is safe for a given INTENDED USE in a given operating environment.”

Safety cases are comprised of claims, arguments and evidence. Several references [4] [5] describe the development and structure of safety cases. There will not be ample space to further explore safety cases here, but they are certainly one approach to conveying the “big picture” of overall risk. It is likely that the FDA will soon recommend safety assurance cases as part of the data to be submitted for pre-market notifications. Indeed, in April 2010, the FDA issued a draft guidance for submissions for infusion pumps [6] in which they stated:

“Although assurance cases have not generally been used in the pre-market review of medical devices, they have been used in other industries with safety-critical systems (e.g., nuclear and avionics). FDA believes the methodology will be particularly useful for presenting and reviewing information about infusion pumps.”

It seems likely that this opinion will soon spread to devices other than infusion pumps. Regardless, safety assurance cases look promising in their ability to support a system level claim of safety, and still be useful for detailed analysis and design input. It is well worth researching safety cases and their applicability to individual devices and project environments.

### Occam’s Razor

You may be familiar with Occam’s razor. This is a principle dating back to the 14th century which has been paraphrased as “whenever there are competing explanations for something, usually the simplest explanation is the most likely.” If you are interested, and are unfamiliar with Occam’s razor, it is worth reading a few references on it. One could argue that the success of relying on Occam’s razor for decision-making, hypothesis testing, and analyzing competing explanations is that it eliminates the more complicated alternatives (hence, the “razor” shaving or cutting them away) and it focuses the mind on the most basic question of the issue at hand rather than being confused by the peripheral complexities of alternatives.

In the context of our overall residual risk analysis, we will twist Occam’s razor for our own purposes.

Occam's razor as it relates to overall residual risk evaluation can be stated as:

"If, when all the residual risks of a particular harm are collectively analyzed, you would not want a stranger to subject your sick mother to those risks (i.e. the simplest, gut level evaluation), then the overall risk is unacceptable—regardless of what other quantitative or qualitative analyses conclude."

Of course this is somewhat tongue in cheek, but a good dose of Occam's razor should be applied to all risk evaluations. If the estimates, scalings, probabilities, regions of acceptability, and so forth lead you to a conclusion that does not agree with your gut, then Occam's razor rules!

### *Tools*

In the discussion of TIR32 above, it was mentioned that risk management takes more than a onetime effort at the end of the development life cycle. It is iterative throughout the life cycle, and requires a variety of tools and techniques. In fact, the 14971 standard suggests that risk management takes:

...more than one risk analysis technique, and sometimes the use of complementary techniques, are needed to complete a comprehensive analysis.

All too often, with all too many of our clients, a single FMEA table is the only artifact of the risk management activity. That is probably because it was the only activity. There are probably good historical reasons for this since the industry's thinking about risk management has evolved over time, and at one time FMEA's were the best and perhaps only tool available.

However, thinking has evolved, and more techniques and tools are available. Today we recognize that certain hazards and risks of harm can be identified and documented on the first day of a development project. This is usually referred to as a preliminary hazard analysis (PHA), and is based on experience with similar products and good engineering common sense. Simple? Yes, but it is highly effective in finding a large percentage of the risk control measures that will be required for the device, very early in the life cycle when it is easy to budget and schedule them in.

Fault tree analysis (FTA) is another excellent technique, and there are a number of software tools to facilitate this analysis. A fault tree starts with the harms possible with a given system, and predicts sequences of events, or faults, that could lead to the harms. Initially, this is purely predictive. As the design evolves it becomes more detailed and leverages predictions based on the actual design. FTAs are a top-down analysis starting with harm working down to specific faults or failures in the system. Compare this to FMEAs, which are bottom-up. FMEAs ask, "What happens if this widget (or software calculation) fails?" In other words, the design objects must be known as well as their relationships to the overall system. An FMEA cannot be done effectively too early in the life cycle, simply because the inventory of items to be analyzed simply are not yet known.

Of course, FMEAs and FTAs both attempt to deal with probabilities. All of our caveats about probability apply to these techniques as well.

Other methodologies that may be found useful are hazard and operability study (HAZOP), and hazard analysis and critical control point (HACCP). Evaluating risk

management techniques is beyond our scope here, but learning more about a variety of techniques is valuable for understanding other ways of thinking about the problem, and some techniques work better than others for specific situations.

It has been noted several times that risk management is iterative and should take place throughout the life cycle. An example of a risk management staircase diagram is provided on the accompanying CD that proposes a number of risk-related activities throughout the life cycle.

It certainly is a good idea to use a number of methodologies and tools for the risk management process. However, one must consider how the information generated from a number of different methodologies will be pulled together for a review that will allow for meaningful overall residual risk assessment. Again, this is beyond the scope of this text but a risk management file with a collection of outputs from an assortment of methodologies does not give one confidence that the full risk picture is well understood unless the information is compiled in one place.

## Summary

This chapter started with a discussion about risk-based validation. The whole reason for discussing risk management to this level of depth is to provide an understanding of the underlying fundamentals of risk management so that the results of the risk management process can be used to fine tune a validation process to minimize safety risk in a medical device.

In the discussion of qualitative probability of software failure, specific factors that modified the probability of failure were identified in an attempt to categorize the probability/risk level. In identifying the specific factors that increase the probability of failure, we were pointed directly to control measures that would reduce the probability. Isn't this what one might call risk-based validation?

Furthermore, in the process of identifying risk control measures for system risks of any origin, software-based risk control measures may have been identified. Those control measures would initially be embedded as safety requirements in the software requirements for the device, and would ultimately be designed, implemented, and tested. Again, would we not call this risk-based validation?

Finally, the issue of probability of software failure was given what some might think was a disproportionate number of pages of coverage. This was done for a number of reasons, but the main two reasons were these:

1. An understanding of how quantitative probabilities combine in risk analysis will help us understand similar issues with qualitative values of probabilities. Qualitative values for probability are all that we really have available for software.
2. It is important to understand that we can make very logical decisions based on only qualitative estimates of probabilities. There is no reason to be distracted by arguments over whether the probability of a given software error is  $10^{-3}$  or  $10^{-6}$ . The argument seems somewhat moot when the failure occurs in the field.

As humans, we need to try to define a process that everyone can follow to produce consistent analyses and decisions. However, at the end of the analysis, the modified form of Occam's razor proposed above must apply: if we would not expose our sick mother to the risks as we understand them, they are simply unacceptable.

## References

- [1] ISO 14971, *Medical Devices—Application of Risk Management to Medical Devices, Second Edition*, 2007-03-01.
- [2] *General Principles of Software Validation*; Final Guidance for Industry and FDA Staff, U.S. Food and Drug Administration, Center for Devices and Radiological Health, January 11, 2002.
- [3] IEC/TR 80002-1:2009: Medical device software—Part 1: Guidance on the application of ISO 14971 to medical device software.
- [4] Kelly, T., *Arguing Safety—A Systematic Approach to Managing Safety Cases*, Ph.D. Dissertation, University of York, U.K., 1998
- [5] Weinstock, C.B. and Goodenough, J.B., “Towards an Assurance Case Practice for Medical Devices,” Carnegie Mellon Software Engineering Institute, October 2009.
- [6] Guidance for Industry and FDA Staff—Total Product Life Cycle: Infusion Pump—Premarket Notification [510(k)] Submissions—DRAFT GUIDANCE, April 23, 2010.



# Other Supporting Activities: Planning, Reviews, Configuration Management, and Defect Management

## Planning

Planning gets a bad rap. Development and validation teams often are not excited about writing plans, and all too often when those plans are written they are filed away and forgotten until the end of the project. At that time they are pulled out and reviewed to see just how close the actual project came to the plan. Nobody is surprised that the plan bears little semblance to the reality of how the project unfolded—sometimes even within the first few days of the project. Often the “plans” are written or rewritten at the end of the project to match the project history. And some people feel this is a waste of time. It is! It is a waste of time because the plan is not being used as a plan. So let us start from the beginning.

Does the FDA think that planning is important in system and software development and validation? Remember the nine design control regulation topics? The first of these is design and development planning.

### Design and Development Planning

Each manufacturer shall establish and maintain plans that describe or reference the design and development activities and define responsibility for implementation. The plans shall identify and describe the interfaces with different groups or activities that provide, or result in, input to the design and development process. The plans shall be reviewed, updated, and approved as design and development evolves.

—21 CFR 820.30 (b)

Let us examine the regulation on our own before looking at how the GPSV interprets the regulation. As with all of the design control regulations, the wording is dense and full of meaning. One can be sure that each word was given careful consideration before including it in this or any regulation. Looking at some of the key phrases:

*Establish and maintain.* The plans are to be maintained! The scenario described above describes a company that stays in compliance with the regulation by “maintaining” it at the end of the development cycle, but have they complied with the

spirit of this part of the regulation? Furthermore, do they get any value out of the plan at all?

*Describe or reference.* The plan describes or references the activities required to achieve the objective. Recall from Chapter 4 that it was suggested quality systems could be structured in such a way that multiple procedures could exist to define the same activity. The plan is what pulls together existing procedures to define a process (such as software development or software testing). A plan does not have to be a volume that describes everything down to the procedure level. In fact, it is preferable that a plan simply identifies the activities, and references procedures that externally describe the activities in detail. Of course, sometimes procedures do not already exist that adequately describe the activity in the context of the current project. In those cases, the plan simply identifies the need for a referenced procedure that describes the activity in detail.

*Responsibility.* A plan has to be more than a detailed list of activities. Without assignment of responsibility, the likelihood of those activities being completed is greatly diminished. An important part of the plan is assignment of roles and responsibilities.

*Interfaces.* Large projects are seldom completed successfully if each of the participating disciplines works in a vacuum. Plans need to identify how the process or discipline being described by the plan interfaces with other disciplines that participate in the project. Often, those responsible for other discipline's plans are made reviewers of each other's plans so that they are aware of the potential interfacing inputs to their own process. Of course, simply listing an interface is not as good as specifying what is to be done to satisfy that need for information from the interface.

*Updated and approved as design and development evolves.* Not only are plans to be "established and maintained" they are to be updated "as design and development evolves." Why is this part of the regulation? It is recognition that plans are never perfect on first writing; they must evolve. It is part of the regulation to be sure that planning is done in a useful way that works to control the design of the product—not in a way that simply documents the historical evolution retrospectively.

Given that detail on what the regulation requires, let us examine *why* planning might be considered an important element of design control, and ultimately validation.

## Why Planning Is Important

Section B of the Design Control Guidance [1] provides some insight into the FDA's rationale for making planning the first of the design control regulations. The rationale is paraphrased and summarized as follows:

*Risk.* Much of today's view of design, development, and validation is based on the premise that the activities can be prioritized based on risk. Planning is no exception. Higher-risk activities need to be subjected to more managerial oversight, review activity, and cross-checks in the design control process than those tasks that are considered to be of lower risk, or more routine. Planning for such prioritization helps assure that the process takes place, and that the high-risk activities are identified early enough in the life cycle that they can be given additional attention.

*Schedules.* As mentioned above, schedules are important part of planning because they are one of the most obvious points of interface among the team contributors. In the Design Control Guidance, the FDA identifies a second rationale for the inclusion of schedules in plans:

... scheduling pressures have historically been a contributing factor in many design defects which caused injury. To the extent that good planning can prevent schedule pressures, the potential for design errors is reduced.

*Management awareness.* Closely related to schedules is the awareness that as deadlines near, or even slip, development and validation teams can be placed under tremendous pressure to complete the project in a timely way. Unfortunately, often-times this results in team members cutting corners to achieve the schedule objectives. A properly written plan that clearly articulates the activities, responsibilities, and risks helps responsible management make intelligent decisions related to scheduling trade-offs.

*Eliminating the “oops factor.”* Even modest device projects can involve a number of cross-discipline activities. The coordination and control of these activities often requires considerable effort. Too often, final design reviews uncover a number of design control, activities that were overlooked as the project progressed (the “oops”). The overlooked activities often include validation activities. The result is that companies often choose to retro-document to make it look as though the activity occurred, to retro-rationalize why the activity was not important, or to simply move forward skipping the activity hoping that nobody will notice. Clearly none of these alternatives meet the intent of the regulation that required the activity. Planning for these activities and managing the projects to follow the plan provides some level of assurance that the design control and validation activities will take place, in proper sequence, to deliver the maximum and desired effect of the activity.

## **How Many Plans Are Required?**

Admittedly, this can be a bit confusing. The regulation calls for design and development planning. The requirement is for planning, not for a specific plan. Section 5.2.1 of the GPSV generalizes the terminology to “quality planning” related to software validation activities. In addition to recommending some of the contents of plans that result from quality planning, this section also provides guidance on specific “tasks” that are typical of quality planning. Included among these tasks are:

- Risk management plans
- Configuration management plans
- Software quality assurance plan, which includes:
  - Software verification and validation plan
  - Schedule and resource allocation for completing the activities
  - Reporting requirements
  - Design review requirements
- Problem reporting and resolution procedures

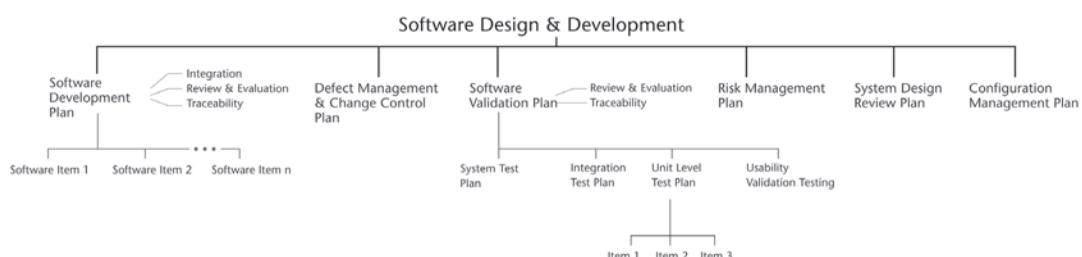
Remember, of course, this list originated in the GPSV, which is focused on software quality/validation activities. Consequently, missing from this list are any plans related to systems, hardware, and software development.

In an attempt to eliminate some of the recursive explanations in the guidance, Figure 9.1 proposes a structure for software development and validation plans. This is not necessarily exactly what regulatory guidance is recommending; however all of the requirements and recommendations in the regulation and guidance are covered by this proposal it is only the structure that differs. This organization is only one of many ways that plans could be structured. Certainly the number of plans and the complexity of the plan structure will depend on how large and complex the development/validation project is.

Figure 9.1 deals primarily with software, but could easily be extended to incorporate mechanical, electrical, electronic and system-level considerations. The plan structure represents the mainstream development/validation process plans: a software development plan, and the software validation plan. Underneath the development plan, are represented subplans that may be needed for complex systems that have more than one software item.

A software item in this context does not refer to a single function, object or source file. It refers to a large scale item that may require a different type of planning. An example of this is a multiprocessor design. The software for each processor could be very different and require separate planning.

Underneath the validation plan are represented subplans for each of the types of validation test activities to be conducted. To the right of the software development plan and software validation plan are supporting activity topics that are self-contained within development and validation that should be covered in their respective plans. These topics include integration, evaluations, reviews, and traceability.



**Figure 9.1** Software development and validation plan structure.

In addition to the two mainstream plans, four supporting process plans are identified. These plans are the risk management plan, configuration management plan, defect management/change control plan, and system design review plan. These are separated from the mainstream plans because they represent processes that must be heavily supported by the two mainstream activities. It would be difficult to place the responsibility for these for activities only within the development or validation test disciplines. An alternative view of the plan organization would be to place these four supporting activity plans in or under one systems engineering plan that is better suited for dealing with cross-discipline activities.

Can all of these activity plans be included in one large master plan? Of course they can. However, for even modestly complex projects, such a master plan can become very large. Responsibility for the creation and execution of the plan is complicated when multiple disciplines are involved. Further, maintenance and the responsibility for maintenance of a large, multifunctional plan becomes an issue as the project evolves.

How does one decide how many plans are needed and how they are organized? Again, it comes down to size and complexity. When the overhead of creating and reviewing multiple plans seems disproportionate to the amount of time invested in the technical content of the plans, it is probably an indication that the plans could be consolidated. Likewise, if the size of the project (or organization) results in only one or two individuals having responsibility for managing the plan(s), that too is probably an indication that the plans could be condensed along the lines of responsibility.

### **Plan Structure and Content**

Although the regulation is not specific to validation, the GPSV provides further guidance on what the agency's interpretation of this design control regulation is as it regards planning and validation:

The software validation process is defined and controlled through the use of a plan. The software validation plan defines "what" is to be accomplished through the software validation effort. Software validation plans are a significant quality system tool. Software validation plans specify areas such as scope, approach, resources, schedules and the types and extent of activities, tasks and work items.

*—General Principles of Software Validation: Section 4.5 Plans*

This paragraph from the GPSV further explains what was intended by the 820.30(b) regulation as it regards software validation specifically. Software validation plans (and all other quality plans) are simply intended to define "what" is being accomplished for software validation for a given project. The details of "how" the activities are completed are reserved for the procedures that are referenced by the plan.

The GPSV extends the list of specifics that are expected in a software validation plan to include the scope (i.e., what software and what activity is covered by this plan), approach (from a high-level view the logical description of the types of validation activities to be applied, and the rationale for their inclusion), resources and schedules (included because these are the most likely interfaces to other disciplines).

Although the GPSV does not specifically include the assignment of responsibilities or requirements for the establishment and maintenance of the plan, they are clearly necessary for a plan to work and evolve in an organization.

In Section 5.2.1 on quality planning, the GPSV is even more specific on its expectations for design and development planning, which incorporates software quality assurance planning (which, in turn, includes verification and validation):

Design and development planning should culminate in a plan that identifies ... a software life cycle model and associated activities ..., as well as those tasks necessary for each software life cycle activity. The plan should include:

The specific tasks for each life cycle activity;

- Enumeration of important quality factors (e.g., reliability, maintainability, and usability);
- Methods and procedures for each task;
- Task acceptance criteria;
- Criteria for defining and documenting outputs in terms that will allow evaluation of their conformance to input requirements;
- Inputs for each task;
- Outputs from each task;
- Roles, resources, and responsibilities for each task;
- Risks and assumptions;
- Documentation of user needs.

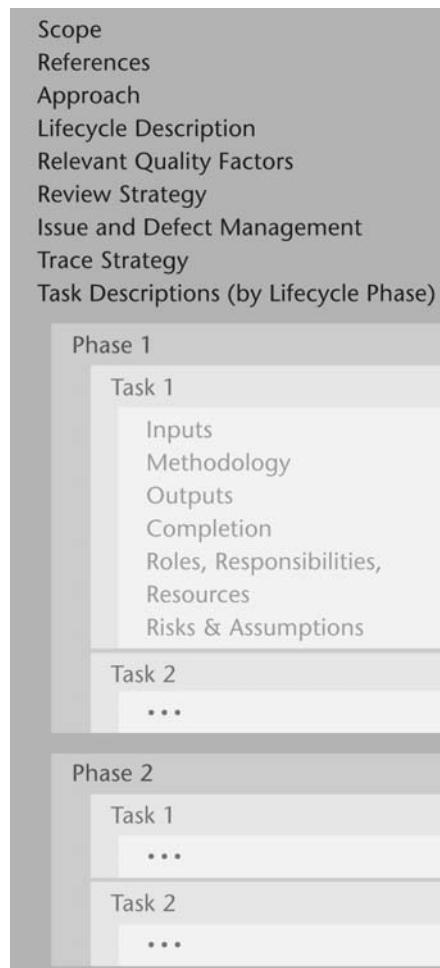
—*General Principles of Software Validation: Section 5.2.1 Quality Planning*

The above excerpt references a life cycle model as part of the planning process. Recall that in the discussion about the activity track life cycle model of Chapter 7, the close relationship between planning and life cycle models was mentioned. If one considers each of the tracks as an activity that corresponds to those activities requiring plans in Figure 9.1, the close relationship between planning, life cycles, and milestones (or “acceptance criteria” in the above citation) becomes more evident.

### What Does a Plan Look Like?

Even with all the above discussion and guidance on planning, it may not be clear what a plan should look like. There is no right or wrong format as long as it incorporates the contents discussed above. The activity track table discussed in Chapter 7 represents much of the content that should be included in a plan. Those who prefer thinking in tabular form may be able to extend that table to incorporate the missing elements of a plan. Alternatively, the missing higher-level elements may be incorporated in a plan that refers to an external activity track table. For those who prefer dealing with text documents, Figure 9.2 proposes a generic outline that can be used as a prototype for design, development, or validation plans.

This generic plan outline is intended for defining the activities related to a process such as software development or software validation. The first eight sections relate to all activities for all phases of the plan. The ninth section breaks the process down according to the life cycle selected. Each phase of the life cycle identifies activities that are to be undertaken within that phase to advance the project. Within



**Figure 9.2** Generic plan outline.

each activity for each phase are the details of how that activity is to be accomplished within the phase, and a description of the completion criteria for determining when the activity is complete for monitoring the readiness to progress to the next phase.

Still referring to Figure 9.2, the following offers some insight into what should and should not be included in each of the sections of the plan:

**Scope.** The scope can and should be kept short and simple, and yet should be useful. The scope should include a description of the process or portion of a process that is covered by the plan. It should also include a description of the item or portion of the item that is covered. For example, a software validation test plan may only cover integration and unit level testing for one processor of several in the system.

**References.** The references should be restricted to referring to documents (such as parent or master documents) that refer to the existence of this plan, or documents that are specifically referred to by this plan. Resist the temptation to load the refer-

ence section with references to all regulatory guidances, standards, company quality system documents, and so forth. Unless there is a specific reference to a document included in the plan, there is little reason to list it as a reference here.

When a document *is referenced*, include the version of that document in the specific reference. Resist the temptation to list “current version” as the reference. That may be meaningful on the day the document is written, but will be meaningless in the future. Admittedly, maintaining version numbers in the reference list will take some work. However, this is part of configuration management. If a referenced document changes version, it is likely that the referring document no longer totally agrees with the reference. Listing “current version” invites problems with document consistency.

*Approach.* This section too, should be kept short and simple. Its purpose is to give some high-level understanding to the reader of what the detailed activities and tasks that follow are trying to accomplish. For example, a plan that covers system-level software verification testing may want to explain in the approach section that the test procedures are following a three-step design, develop, and execute methodology. That kind of high-level description is hard to reverse engineer from only the detailed outputs that result from the activity.

The approach might also explain what test types are included, and how the tests are partitioned into protocols.

*Life cycle description.* This section simply describes the life cycle model that is chosen for this process. This may be different from the overall system life cycle for the development/validation project. If so, some additional explanation may be required for how this life cycle model interfaces with the overall life cycle of the project. If a preexisting internal life cycle standard, guidance, or procedure exists, it should be referenced rather than describing it in detail in the plan.

*Relevant quality factors.* This is a description of desirable qualities of the outputs of the process being described by the plan. When possible, these factors, which are often more qualitative than quantitative, should be expressed in ways that can be measured or validated. For example, a desirable quality factor for a test plan might be to specify what percentage of the code is to be covered by the test activities described in the plan. Alternatively, this section could be used to set goals for special treatment of software that is determined to be related to high risk. In general, this section should be used to provide some verifiable understanding of how the quality of the output is to be judged.

*Review strategy.* If the evaluations and reviews of activity outputs are not covered as individual tasks in the detailed planning section, then the strategy for how the outputs will be reviewed should be covered here.

*Issue management.* If issues discovered during the execution of this plan are not captured in an overall project defect/issue tracking plan, then the details for how issues are to be tracked in this self-contained set of activities needs to be discussed in

the section. Alternatively, this section could refer to a preexisting procedure for handling issue management.

*Traceability strategy.* Any traceability activities that are incorporated in this plan should, at a minimum, be described at a high level in this section. If the details of the trace activities, tasks, and outputs are not included in the detailed planning phase below, then the details belong in this section. Alternatively, if the strategy to be used for the project is similar enough to other past projects, then if those traceability procedures are already documented in an external procedure they could be referenced in this section. Regardless, the reader should come away from this section with an understanding of what traces to expect, where to find them, and what level of granularity they capture.

Inside the detailed plan, the activities are divided into the phases of the selected life cycle model.

Within each activity of each phase, the following items should exist:

*Description of tasks.* This is a high-level description of the tasks to be included within this activity track for the phase being described. The purpose is to give a high-level view to assist the reader in understanding how the various tasks within the activity may be interrelated (or not).

For each task:

- *Inputs:* A description of the inputs that are necessary to complete the task being described. Since this is likely to identify interfaces to other processes, the source of the input should also be noted.
- *Methodology:* This is a description of the work to be done for this task. Some level of detail is to be provided for how the work item is to be completed. Alternatively a reference to an external procedure describing the work could be referenced. Since this plan is a working plan, enough detail should be provided so that coworkers who are charged with following the plan will have adequate explanation not only of what they should be doing, but of how they should do it.
- *Outputs:* These are the outputs that results from the work described in the methodology. In some cases the output may be a completed document, or software item. In life cycle models that allow phased releases of deliverables, this section should be used to define the goals for the completion level of the for this phase. For example, if one of the outputs for an activity is the first release of a requirements document, the output section might describe the “Release 1” document with certain sections completed. In the next phase, the second release of that requirements document would be described in the output section for that activity track.
- *Completion criteria:* For a specific task, this section describes the expectations for completion of a specified output. The completion criteria for all tasks in a given phase can be collected together as a milestone description document for determining the readiness to progress from one phase of the life cycle to the next.

*Roles, responsibilities, and resources (the 3 Rs).* Each task in a plan should be assigned to a single individual who will be responsible for its completion. Any number of people may participate in the completion of a task, and their roles are described here. Resources needed to complete the task should include human resources, tools, access to prototype equipment, and access to outside resources.

Although the 3 Rs need to be planned on a task-by-task basis, it may be a better summary view of the roles and responsibilities to use the RASCI charts that were described in Chapter 4. If that is done, the section simply describes resources and refers to the RASCI chart for roles and responsibilities.

*Risks and assumptions.* These are not risks related to the safety of the device being developed or validated, but project risks and project assumptions. Why is this necessary? Especially for projects that involve more than one plan, specifically identifying the risks and assumptions related to completion of the given task give management a view of the likelihood of achieving completion of the task on schedule. Assumptions that are not realistic can be identified quickly before too much time is wasted planning around them. Project risks that could jeopardize the success of the project or the schedule can be identified early to give management and opportunity to find ways to reduce the risks. What value is there in this? As mentioned above under scheduling, schedule pressures are often associated with device defects. Any management techniques that can be used to reduce schedule pressures, such as identifying project risks and reducing the risks, theoretically should result in better quality outputs.

## Evolving the Plan

In several places in the regulation and guidance documents, the establishment and maintenance of design and development plans were referenced. One of the major reasons given for the reluctance to create such plans is the fear that the development or validation team could be held accountable for detail in a plan that is later found to be more difficult than anticipated. There is also, frequently, a feeling that creating plans is a waste of time since no one can anticipate the details of how a project will unfold once it gets underway.

There is no expectation in any regulation or guidance document that highly detailed plans created on the first day of the project, can be followed to the letter through to the completion of the project. In fact, the regulation itself (21 CFR 820.30-b) states, “The plans shall be reviewed, updated, and approved as design and development evolves.”

This seems to be a very reasonable requirement that counters many of the perceived objections to planning. The acceptability of evolutionary planning also allows one to plan only as far in the future as one’s vision of the process is clear. There is certainly nothing wrong with later phases of the plan having very little detail in them at the early stages of the project. As the project moves through the life cycle phases, additional detail should be added to those later phases as the ability to predict what needs to be done in those later phases improves with increased information available.

## Configuration Management

Configuration management (CM) and configuration management plans (CMPs) are easily overlooked by the inexperienced developer and validation engineer. The need for configuration management planning, if not understood early in a project's quality planning, will certainly be understood once the activities get underway. Unfortunately, at that time, it is very difficult to "get the genie back in the bottle" and effect controls mid-project. Among the software engineering (developer) population, CM is often thought of synonymously with version control. Certainly version control is a central part of CM, but as will be suggested shortly, CM involves quite a bit more than version control.

### Regulatory Background

The FDA's General Principles of Software Validation (GPSV) provides only a little guidance on configuration management planning, and all of that falls under the section covering quality planning:

A configuration management plan should be developed that will guide and control multiple parallel development activities and ensure proper communications and documentation.... Controls are necessary to ensure positive and correct correspondence among all approved versions of the specifications documents, source code, object code, and test suites that comprise a software system.

—*General Principles of Software Validation: Section 5.2.1*

Certainly the coordination of activities from multiple parallel efforts has become a significant need as the complexity and size of device software has increased by orders of magnitude since the 1980s. The parallel activities may be multiple software development activities with different teams. Those teams may be focused on different aspects of the software system design (as in multiprocessor designs), or may be multiple teams working on the same software, and often they are geographically dispersed. Communication and documentation are key to the coordination of efforts. Keeping the communications and documentation in sync with the realities of the state of the design outputs is the objective of configuration management.

Although the GPSV is focused on software systems, most device software is embedded in some kind of electronic or electromechanical system. Coordination of communications between the software team and the hardware team(s) is also a key objective of CM.

Finally, coordination of communications and documentation is critically important to the verification and validation team. These team members, when working in parallel with the development effort, only have communications and documentation (hopefully more documentation than verbal communication) to use as inputs to their efforts. Keeping these inputs synchronized with the actual development effort is necessary for the test suites to be relevant to the design outputs once they are ready for test.

The controls also should ensure accurate identification of, and access to, the currently approved versions.

*—General Principles of Software Validation, Section 5.2.1*

This recommendation is easily handled by almost any of the many the version control systems that are commercially available. This is what is often confused as the main requirement for CM. The two references to “approved versions” in the quotations from the guidance should alert the reader that perhaps more is intended with CM than simple version control check-in/check-out procedures.

### **Why Configuration Management?**

In the “good old days” when medical device software was of limited size and complexity and was usually written by one or two software engineers (often sharing an office), documentation and configuration management required less formality. Communications were often verbal, or left on a whiteboard or on scraps of paper. It was easy to keep the one or two developers in sync. Of course the needs for documentation for long-term maintenance existed, but documentation was often given only superficial attention because it was assumed the programmers would always be available for maintenance issues. Version control was on the honor system, expecting the developer(s) to at least keep separate floppy disks or directories for each version. The developer made his or her own decisions about what changes to implement, and when to implement them. Change control was often a foreign term, or what the hardware guys had to do because they were unfortunate enough to have to document changes for production. As times have changed, the need for documentation, change control, and the management of configurations has become evident. However, it is because of the past from which device software has evolved that there is to this day so much resistance to documentation, change control, and configuration management.

Perhaps the best way to describe why CM is important is to imagine the chaos that might exist if there were no configuration management planning as in the following example.

Imaginary Company X spent hundreds of thousands of dollars in defining their new product. A well-written system requirements specification (SyRS) was created that pulled together the needs of all the stakeholders. Users, decision makers, marketing specialists, manufacturing, and human factors experts all were consulted at great expense to define the “right product.” Software requirements were started but the user interface requirements from the SyRS were abandoned because new technology allowed for nice new icons—even animations. The new technology was panned into the software requirements specification (SRS), ignoring that the software requirements no longer fulfilled the product requirements set forth in the SyRS. The developers were unaware that the human factors experts had done a detailed study to determine how much user confusion would be created if their customers had two user interface styles to deal with in the same clinical setting for two devices from the same company.

The software design description (SDD) was started, and the user interface requirements of the SRS were now ignored because a new development tool would

speed development, but it was not capable of some of the new features detailed in the SRS. The implementers started creating code, but deviated from the SDD because it was “too time-consuming” to think through the design details of the common modules that controlled the functionality of the user interface, so each developer was allowed to create that functionality from scratch. Meanwhile, from the SRS, the system-level verification test team kicked off their creation of the system-level software test protocols. Of course, the SRS no longer matched either the documented design or the implemented code.

The first indication that something was wrong was when the trial runs of the test protocols started identifying large numbers of defects. After consulting the developers they were told that the SRS was not followed “to the letter”; probably the SDD was a better indication of what was really implemented. Rewriting the test protocols to match the assumed requirements from the design detail uncovered a different set of defects. The implementers admitted they really only read the SDD once, and implemented the source code from their “impressions of the intent” of the SDD. The best information really could be found in the source code itself.

There is no point continuing. Many readers have seen this scenario before. Many who have seen it have seen it repeat itself. Time and money were wasted. There was not a complete and accurate set of documentation at the “completion” of the project. The design and implementation migrated so far from the SyRS requirements that many of the marketing, clinical, and user needs were not met.

A lack of design controls, partially implemented through a configuration management plan, was to blame. Of course, if the quality system didn’t call for a CMP, then that would be the root cause. The simple fact is that the design evolved (though well intentioned) in an uncontrolled way. Parallel activities, though attempted, were not possible because the design inputs were unreliable. The attempted parallelism was frustrating for all concerned, did little to speed time to market, increased project costs by a factor of five, and got absolutely no credit or recognition for the hard work that went into chasing a moving target. Oh, and the product wasn’t the product anyone thought they were getting.

But that wasn’t the end of the problems. Auditors and inspectors arrived and wanted to understand how the version of software being shipped in the product traced to the source code and build files; it appeared that some of the source files were dated more recently than the executable that was shipping. Years later, the development team (90% new members) wanted to add some new features, but could not figure out how to build the executable to match the version being shipped. There was not a good baseline from which to start. The compiler had been updated, and the developers had a hard time even getting a clean compile.

How could a well-constructed CMP have helped? Let us look at what goes into a CMP, then revisit how things might have been different.

### **What Goes into a Configuration Management Plan?**

Since our regulatory guidance is limited to objectives for CM and gives little guidance on how to manage configurations, we’ll turn to an industry standard on the topic, the IEEE Standard for Software Configuration Management Plans (IEEE Std 828-1998).

The first thing to point out is that the CMP *is a plan* and should contain all the components of a plan that were previously mentioned under the planning topic. We won't repeat them here, but that doesn't mean they are not components of a CMP. Obviously, the activities related to CM are unique, and that is where we will focus attention here.

The IEEE standard is very thorough and is a good reference on this topic. However, leaving the fine points for the standard, the main topics to be covered in a CMP are discussed here.

### Itemization of the Configuration Items (CIs)

Identify the pieces and parts that will comprise the system. This includes the software outputs (source code, binaries, libraries, off-the-shelf components, build files, executables) for each software item in the device. A device may be comprised of numerous software items depending on the physical architecture (how many processors) and how many independent software subsystems exist.

Often overlooked are CIs that are included in the finished product that were not custom developed, but are necessary for the proper operation of the device. Included among these are operating systems, libraries (included by compilers), drivers, and external software such as communication stacks, and graphics functions.

Additionally, all documentation such as requirements, designs, applicable coding standards, and test protocols should be listed as configurable items. Any output, whether software or documentation, that is created or included for the benefit of the finished product, should be itemized so that the overall configuration can be understood, managed, and documented for posterity.

Since the CMP is a plan, one might consider keeping the actual itemizations or configuration definition outside the plan itself but referenced by the plan. Some type of controls should be placed on the configuration definition (i.e., who decides to change it, who is responsible for maintenance, version history requirements, etc.).

How much detail or resolution is needed? A lot. Source code resolution should be at the source file level (no need to identify a version for each function or object). Other than that, every item that is needed to build the software (including the tools, compiler, and build instructions themselves), test the software, or describe the software should be included. Think to the future. In a future maintenance project on the software, what will you need to create the environment to recreate a bit-for-bit identical version of the released version of software? What documentation will you need to rely on to describe the intent (requirements), design, and means for testing the software?

### Itemization of Any External Software Tools Used

Any tools used in the design, development, or control of the developed software should also be considered part of the software configuration. To consider an obvious example, if a compiler is upgraded to a new version, it is likely that the compiled version of the device software will change from one compiler version to the next. Such a change in configuration could invalidate the entire set of test results gathered from versions of the software compiled by the older compiler. This should provide

some insight into the level of detail that should be monitored in managing configurations. If a compiler upgrade did take place mid-project, would you have enough information to decide what tests would have to be repeated because of the compiler change?

### Version Control and Naming of Configuration Items

Naming requirements need to specify how one version of a CI will be uniquely recognized as being different from the same CI of a different version. This can be handled by tools, file naming conventions, time/date stamps, or comments in source headers.

Version control (or revision control, source control, etc.) is what software developers often confuse with configuration management, because there are good tools to assist with it. Version control involves the coordination of multiple developers who need access to common source modules. This is accomplished in automated tools with check-out and check-in procedures. Modules that are checked out cannot be modified until checked back in. More sophisticated version control tools allow for version merging. This allows multiple developers to have the same file checked out at the same time, and as the files are checked back in there are features that allow for merging the versions together so that the improvements of one developer are not lost when the second developer introduces a change.

Version control, like almost everything that has to do with software, can get quite complicated. This is not the place to get into the fine points of version control, but one can appreciate that concepts like branching and common elements used to support multiple devices or multiple versions of the same device can add complexity to version control.

One of the key concepts of configuration management is that of the baseline. Baselines will be discussed shortly under the topic of change control, but the definition and management of baselines is intimately intertwined with version control and needs to be discussed to some extent here as well.

Imagine at a given point in time one wants to understand the status of the entire software development/validation project. If one takes the current version of every CI in the configuration definition list at that point in time, that would (loosely) describe a baseline of the code at that point in time. Version control needs to support baseline management by providing a level of detail about the version status of each CI at a baseline point. If the project team at some point in the future needs to return to a baseline, the version control system must support the storage and retrieval requirements necessary to define and re-create a given baseline.

### Storage Details

These may seem like simple details, but they will not be to new members of the project team, and will not be to anyone several months or years after the project is completed. Issues to be detailed under this topic include:

- Where the CIs will be stored;
- How the CIs will be accessed;

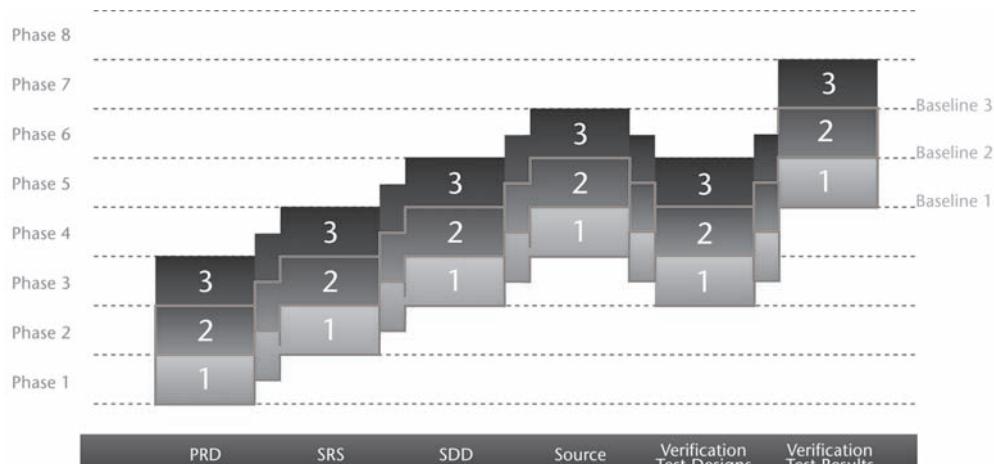
- How access will be controlled;
- How the CI repository will be backed up while the project is active;
- Where the CI repository will be archived once the project is complete;
- Identification of procedures for restoring the repository from backups or archives;
- If software or documentation is bundled into libraries or master documents, a description of the requirements for supporting that process must be described (software to be used, file locations, procedures, etc.).

### Change Control

The previous items found in the CMP are, for the most part, focused on documentation of procedures to be used. Perhaps the most important element of configuration management is the change control element in which the management of the configuration is described.

Previously, baselines were defined technically. A more interesting and difficult question is to consider how often a baseline should be defined, and what should go in it. Thinking back to the activity track life cycle model described in Chapter 7 might give us some insight into how baselines could be managed.

The life cycle phases under the activity track model detailed activities on a staggered basis. Designers and testers were working on tests from the last approved version of requirements, while requirements were continuing to be refined. Figure 9.3 shows a diagrammatic view of the relationship between life cycle phases (using the activity track model) and baselines. Note that the heavy baseline staircase marks out a version of all documents and software that relate to the same version. This, of course, assumes a version control tool that allows for baselining for historical versions of certain documents, while allowing progress to move forward. For example, the full baseline for Baseline 1 cannot be completed until Phase 4, but by Phase 4, the SRS is on version 3.



**Figure 9.3** Relationship of activity track life cycle phases to baselines.

Understandably, there are always issues that keep all of the outputs from being perfectly in agreement. There is no reason a baseline cannot be defined with the assumption that all outputs agree with each other with the exception of those listed in an exception list.

Management of the baselines should be defined in the CMP. Often, responsibility of change management is given to a committee, sometimes called the change control board (CCB). Regardless of who was given this responsibility, those responsible should be deciding what new features, fixes, and changes find their way into the configuration from version to version and from baseline to baseline. Many project CMPs define this responsibility in a way that gets more formal and strict as the project progresses and nears release. Certainly, in later phases of the project, tight controls are needed to be sure that simple changes do not result in disastrous functional, budgetary, and scheduling results because of unanticipated effects on other parts of the configuration.

The change control process should include:

- A formal identification of which CIs change from version to version and baseline to baseline;
- A description of the change approval process, and whose responsibility it is.
- Allowance for verification of any changes that are made to the configuration. For example, consider a change to fix a defect. One would want verification that the defect was actually fixed;

### Coordinating Configuration with Externally Controlled Items

Since medical device software is usually part of some electrical or electromechanical system, one cannot control the configuration of the software without consideration of the rest of the system. The software's CMP should address how the configuration of the overall system and that of the individual non-software subsystems is to be coordinated.

### Control of Externally Developed Configuration Items

Given the size and complexity of modern medical device software, it is quite common for some software development or validation activities to be outsourced or codeveloped with other team members who are not geographically coresident. In crafting a CMP for device software, even externally developed software should be considered as part of the configuration definition. Procedures for coordinating the management of the configuration between the internal and external groups should also be covered in the CMP. Special challenges may be presented when working with external groups working under their own CMPs and/or version control systems.

Returning to our short example above, a well-thought-out CMP would have defined baselines of the software "configuration" rather than allowing the SDD and source code to evolve at their own pace.

As with the other topics covered under the supporting processes chapters, configuration management, too, is deserving of a book of its own. It is an important

aspect of medical device software development and validation that has direct ties back to the regulations on change management. More is involved than simple version control using a software tool. Configuration management is also very much related to project management, and life cycle management. It defines the control points (i.e., baselines) around which the design and validation of the product are managed.

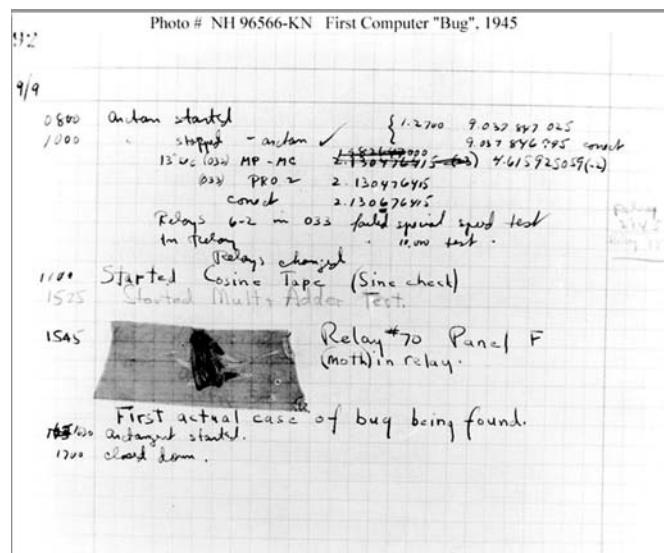
## Defect (and Issue) Management

Problems, defects, anomalies, bugs, and issues are all terms used to describe concepts that are either synonymous or closely related. The “bugs” terminology dates back to the Mark II computer in the 1940s when a moth was found trapped in a relay of the computer (see Figure 9.4). The moth (bug) in the relay resulted in unexpected results from the computer. “Bugs” in modern usage are generally thought of as flaws in software. The term is commonly used, but is considered an informal term.

The term “problem” generally refers to a problem report from users, testers, or reviewers about troubles with the device or more specifically in our context, the device software. The problems may relate to perfectly engineered device characteristics (i.e., not bugs), but the wrong device characteristics.

Defects and anomalies are usually treated as synonymous terms, are rooted in regulatory terminology, and refer to device flaws that require repair or sufficient analysis to legitimize their disposition without repair.

We have found it useful to introduce the term “issues” to track concerns that come up during the course of a product development life cycle. Issues deserve to be tracked and managed, but are not technically defects or anomalies in a *finished*



**Figure 9.4** Lab notes documenting the first computer bug (from <http://www.history.navy.mil/photos/images/h96000/h96566k.jpg>).

*implementation* of the device. Issues often are raised during reviews of requirements and designs and may be reminders for missing requirements, incorrect requirements, conflicting requirements, and so forth. We have found it extremely useful to begin tracking issues early in the development life cycle. As engineers we are paid to review documents and find the issues. Shouldn't we keep track of these valuable assets (yes, issues are assets) to be sure our employers or clients don't have to pay us to find them again later when it is more costly to resolve them?

## Regulatory Background

The FDA Quality System Regulations (QSRs) do not contain any references to defect management. The GPSV, under the topic of Quality Planning (Section 5.2.1) says:

Procedures should be created for reporting and resolving software anomalies found through validation or other activities. Management should identify the reports and specify the contents, format, and responsible organizational elements for each report.

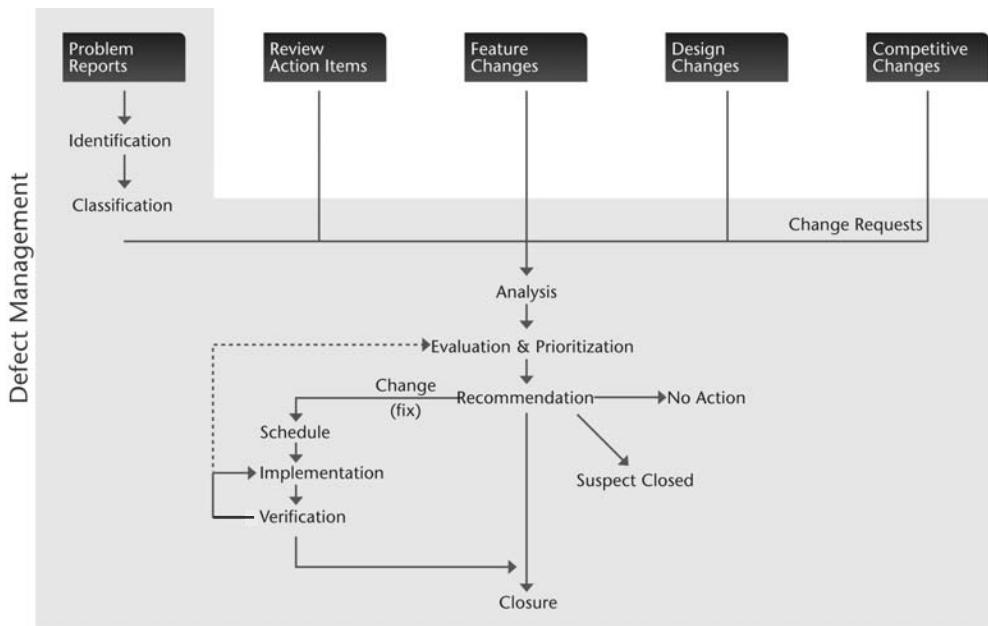
Since this is under the topic of quality planning, it refers to suggestions for a defect (anomaly) management plan. As said numerous times before, one of the key ingredients of a plan is the assignment of responsibility. Defect management plans are no different. If no single person is singly responsible for dealing with this important function, it is almost guaranteed that it will not be taken seriously. Since defect management and change management are so closely intertwined, the defect management plan is often included in the configuration management plan that details change management.

## Why Defect Management Plans and Procedures Are Important

Hopefully, there is no disagreement that serious and potentially dangerous defects should be repaired as quickly as possible. A systematic defect management plan and procedure makes it more likely that the right information is collected to support sufficient analysis and identification of root causes to determine which defects are important and which are not.

## Relationship to Configuration (Change) Management

Change management procedures and defect management procedures are very closely related. A defect can certainly be managed within a generic change management procedure so long as those aspects that are specific to defects are singled out and handled separately. (Realize, of course, that not all changes will be as a result of defects.) Figure 9.5 shows one view of how changes to a device can be managed. Note that change requests that originate from problem (defect) reports take a separate path initially because there are certain activities related to identification and classification that apply to defects that are not as applicable to other types of change requests. The regulatory guidance quoted above also suggests that procedures and reports related to problem reporting may need a level of formalization that is some-



**Figure 9.5** Defect management as part of change management.

what more rigorous than other change requests. This is especially true once the device has been released and is in the hands of actual users. Once the problems are identified and classified, the changes (i.e., fixes) needed to correct the problems can be analyzed, evaluated, and prioritized like any other change request.

A problem report is a strong indication, but not a certainty, that something is wrong with the device. A test has failed, someone has noticed that something did not behave in an expected manner, or a new hazard has been identified that must be addressed. Something needs to be done to right the wrong. There is a potential for a negative impact on safety. All this implies a higher level of urgency than a change that someone has requested to implement a minor improvement.

The pathway from report to closure includes a number of activities, each of which is described below. Some consideration should be given to how these activities are documented. Remember that in the eyes of an inspector or auditor, if it is not documented then it did not happen. Fortunately, there are numerous defect tracking software systems available that are excellent at documenting the thought process, and tracking the progress of defects or other changes from identification to closure. The defect management procedure may contain the following activities:

**Identification.** The suspected defect is found, a problem report is prepared with detailed descriptions of how it was made to occur so that others can recreate it.

**Classification.** Not all defects are simply software errors. It could be a misunderstanding on the part of the person reporting the defect (normal behavior). It could be that the software is correct, but the requirement and the related verification test are wrong (defective requirement). It might be unexpected or undocumented behavior (missing requirement). It could be a duplicate of a previously reported defect, or it

could be another manifestation of the root cause of a previously reported defect. Regardless, the classification is important for later determination on the recommendations and means for verifying any changes made to deal with the defect. Classification is usually handled by the more senior members of the defect management team, or someone with detailed knowledge of the software (device) requirements and day to day knowledge of the defect database.

*Analysis.* The defect may or may not have a significant impact on the functionality, safety, marketability, or clinical effectiveness of the device. The analysis makes this determination and is a major input for determining the criticality and urgency of fixing the defect. Analysis sometimes needs to be handed back to the development team for additional information if the details of the defect report are few, or the understanding of the defect is otherwise limited.

*Evaluation and prioritization.* The classification and analysis are evaluated to determine the priority of the needed repair; that is, the need to repair the defect (criticality) and how soon that repair should be implemented (urgency). Urgency could be dictated by safety concerns (for example if the software is already released for human use, or simply could injure the testers). Urgency may be determined by project needs (the defect is keeping development and/or testing from progressing downstream). The evaluation considers possible solutions to the defect, and trades off the relative merits of alternate solutions if they exist. The relative merits include not only how well the defect is repaired in the software, but also the impact on the entire documentation hierarchy, test protocol design, and retesting that may be required by the software fix.

*Recommendation.* A recommendation considers the priority, impact analysis, and evaluation of alternative solutions to recommend the best alternative. Inputs from the development team, test team, clinical and/or marketing specialists may be polled for especially difficult decisions. The recommendations generally fall into one of four major categories (which could be further subdivided if that adds any value). Those four categories are:

1. *Implement the change (i.e., fix the problem):* The recommendation should provide enough detail so that it will be clear to the implementer how it is to be resolved. Furthermore, if any verification of the change is needed, enough detail should be provided that the verifier will know if the resolution is effective, and as recommended.
2. *No action or deferred action:* The evaluation may conclude that the priority of the defect or other change is so low that it is not worth the cost in time and budget, or the risk of injecting other defects to fix it. In that case the defect is simply left “alive” (i.e., not closed) in the defect management system with the low priority ranking and a recommendation of “no action.” In cases like this, it is better to have some status other than “closed.” Closing the defect would simply create an opportunity for it to be rediscovered, reported, and analyzed over and over again. It is also a good engineering practice that

manufacturers understand any limitations in their devices due to defects that are not repaired—no matter how low the priority.

3. *Close:* A change request is closed when it is implemented and verified, found to be a duplicate of another change request that is not closed, or when the change request is withdrawn for some documented reason. Closure rationale should always be documented so that subsequent reviewers understand whether closure was routine (i.e., fixed and verified) or was somewhat more involved.
4. *Suspect closed:* Every project ends up with a collection of problems that cannot be resolved because they disappeared before the analysis discovered the root cause, or they were observed but could not be replicated. There are legitimate reasons that this happens. Often these problems are discovered relatively early in the life cycle and are simply fixed as a side effect of another fix, or simply as a side effect of the maturation of the software. The developers cannot fix what is not broken, but it is not right to simply declare the problem closed either. Personally, this is the category of problems that I lose a lot of sleep over. I really do not like to have problems I don't understand. However, some problems do end up in this category. A good practice is to factor any “suspected closed” problems into the system risk analysis. If the problem could result in a high severity harm, then it just makes sense that some further analysis may be justified to understand how or why it disappeared, or whether the original problem report may have been in error, or whether it is still there, but efforts to recreate it simply have not yet been successful. Regardless, the wrong thing to do is simply close these and treat them like a legitimately fixed and verified problem. This category has a way of resurfacing once the device is in the field.

*Implementation:* The implementer works from the details provided in the recommendation. If the recommendation is not clear, the implementer is responsible for eliciting enough detail in the recommendation to make it clear. Going ad lib on implementing fixes is not a good thing. All the deliberation of the defect management team on analysis and evaluation is wasted, and the design starts to go out of control if the implementers work from their own script. If the implementer feels he/she needs to ad lib because not enough detail is given, then the analysis/evaluation/recommendation parts of the procedure need remediation.

*Verification:* Assuming the implementation follows the recommendation, and the defect identification and classification provide enough detail, someone other than the implementer should be made responsible for verification of the change. The original person who reported the defect is a natural choice as verifier of the resulting fix, but there is no reason it needs to be that person. What is important is that the responsibility is not given to the implementer himself or herself.

*Closure:* If the implemented change is successfully verified, and all required updates are made to documentation and test procedures, then the defect can be closed. A practice we have used in the past is to include a formal test procedure to test for the occurrence of the original defect into a separate class of test protocols.

Our rationale for this is that we noticed over the years that “bugs” really act like bugs. They have a tendency to come back once you have had them the first time. In previous topics related to the probability of software failure, the defect history of the software unit was cited as a factor that partially determines the likelihood of future failure. Including this “past defects test protocol” collection of test procedures in routine runs of regression tests has proven to be productive in catching defects that somehow creep back into the software.

## Planning for Defect Management

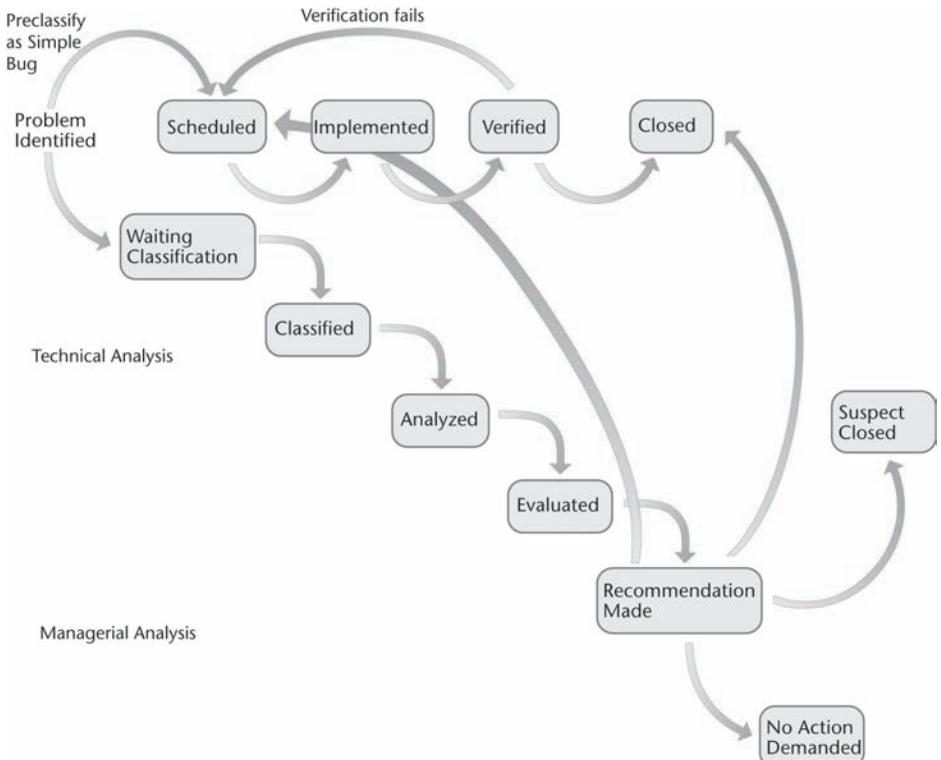
Inexperienced medical device project teams almost always underestimate the importance and magnitude of the defect management activity. Defect management and configuration management are closely related because decisions that are made (or not made) will affect the configuration of the project deliverables. Careful consideration is required in the analysis and evaluation of defects. The evaluation should consider not only how long it will take to modify the software, but also how that software modification will impact the configuration. That is, one must consider how the requirements, designs, test protocols, review status, and so forth would be affected by a simple software change. These ripple effects of a software change are often much larger and time-consuming than the software change itself.

A defect management procedure is shown in Figure 9.6 in a state diagram representation. This is a simplified version of what generally is done for projects of moderate size and larger. It is shown here to make a few points.

During defect management planning, the roles and responsibilities will have to be assigned. One approach is to make defect management (DM) the responsibility of one person or group. Alternatively, different people can be responsible for different parts of DM. Note in the figure how the early states of the procedure are very technical in nature (e.g., is it really a problem? Is it a duplicate? Is it software only, or will other outputs be affected? How serious is it?). Later states in the diagram start to be more managerial in nature (How long will this impact schedule and budget? Is it important enough to fix now? When should it be scheduled to keep all efforts in sync?)

Perhaps one of the worst ways to handle DM (although very common) is to make everything a committee decision through a change control board (CCB). CCBs have their place. They are needed to get the full range of inputs into the decision-making process from all the project stakeholders. However, they do not need to be involved in every decision. By requiring that, one will find that the CCB will soon become the bottleneck in the project. Worse, some decisions will be made by a majority that is perhaps not best qualified to make the decisions. Even worse, if the CCB does become a bottleneck, team members will do what they can to avoid exposure to the CCB to keep the project moving.

In Figure 9.6, notice that a preclassification step was put in place to separate out simple bugs. Of course, one would want the definition of “simple bug” well documented, as well as a clearly defined assignment of responsibility for the person(s) making these decisions. The point to make here is that some problems are simply errors in the software implementation. (Yes, those are the simple situations!) For example, a background is red but should have been green, or a value is



**Figure 9.6** State diagram representation of defect management procedure.

truncated but should have been rounded. The software is only wrong; it just needs to be fixed and reverified. To involve a CCB in a simple nondecision like that is a waste of everyone's time. A good DM plan considers who needs to be involved in each step of the DM procedure, then, includes those needed, but no more.

There is one final caveat. The defect list or database often becomes a target of some corporate manipulation. It is not uncommon for the defect list to be blamed for why the project is behind schedule. It is equally common to see those under schedule pressure try to combine defects or prematurely close defects simply to reduce the count to show progress. When CCBs cannot keep up with the influx of new defects being reported, they have been known to change the classification of what constitutes a defect to try to keep the defect list small. Sometimes, especially early in the life cycle, the team is discouraged from logging new defects or issues simply because the CCB, as organized, cannot keep up with the volume of reports. Does that make sense? Of course not. The problem is not the defect list, it is the procedure and assignment of responsibilities that is not working.

Often, the manipulation of the DM process in general, and the defect list in particular has something to do with the inability of some to deal with the facts that define the state of the project. Project management is always better off finding better ways of dealing with the truth than creating new ways to obscure the truth. A defect is a defect—just deal with it!

## Reviews

Early in my career (actually as a co-op student) while writing operating system software, I was invited to my first review meeting. At the time everything was new and interesting. I was excited about this review. I viewed it as an opportunity to learn about other's design and implementation techniques, and how they were tackling their problems. I even looked forward to contributing by finding problems and bringing some of my freshly learned skills from school to the group. I shared my enthusiasm with one of my closer friends in the group who was a full-time engineer. I questioned why nobody else seemed interested in the review, and why this was the first review I knew about after almost two years working with the group. The only response I got was, "You'll see."

It didn't take me long. After the first 20 minutes of the review, I hoped never to be invited to another one, and prayed that my work would never be reviewed. The review consisted of a software engineer reading his source code to us, followed by hours of criticism, personal comments, pontificating about personal opinions—all wrapped in a generous portion of harsh language. I was stunned. The ten "programmers" in this group were close friends. We played softball together, socialized on weekends, and knew each other's spouses and girlfriends. What in the world caused this behavior? Worst of all, very little was ever done about the comments made at the review. There was no documentation of the findings or what was or wasn't done about them. It seemed that reviews were just some form of perverse software gauntlet that we all had to survive at one time or another. Nobody was sure why we had to do it. Everyone did whatever they could to avoid them.

My experience with reviews in the medical device industry has been somewhat better, but elements of that first review still are present in many of the reviews that I attend with our clients even today. Yet, we have gathered project metrics that support our belief that reviews are among the most productive method of finding software defects and issues. They are productive in terms of the number of issues and defects found, and in terms of the cost per defect found. Since there is great value in reviews, there is ample reason to make improvements to the review activity to maximize the value gained while reducing the pain to tolerable levels.

## Regulatory Background

The requirement for reviews is rooted in FDA regulation and guidance. The design control regulations refer to the need for reviews in two places. In 820.30(c) on the topic of design inputs, the regulation states that device manufacturers should, "... ensure that ... the design requirements relating to a device are appropriate and address the intended use of the device..." and that procedures should exist to "...include a mechanism for addressing incomplete, ambiguous, or conflicting requirements." Although this is a somewhat indirect regulatory requirement for a review, it is clear that design inputs are to be checked for appropriateness for the intended use of the device, and that a mechanism for dealing with any design input deficiencies must exist.

A little more direct is 820.30(b) on the topic of planning, which requires, "The plans shall be reviewed, updated, and approved as the design and development

evolves". Similarly, 820.30(d) on the topic of design outputs requires that, "Design output shall be documented, reviewed, and approved before release," and 820.30(i) on the topic of design changes, which requires "...review, and approval of design changes before their implementation."

There is a lot of reviewing expected by the QSRs. So what is meant by a review? The design control section of the QSRs gets very specific in 820.30(e), which is devoted to the topic of design review:

(e) Design review. Each manufacturer shall establish and maintain procedures to ensure that formal documented reviews of the design results are planned and conducted at appropriate stages of the device's design development. The procedures shall ensure that participants at each design review include representatives of all functions concerned with the design stage being reviewed and an individual(s) who does not have direct responsibility for the design stage being reviewed, as well as any specialists needed. The results of a design review, including identification of the design, the date, and the individual(s) performing the review, shall be documented in the design history file (the DHF).

The underlined words and phrases are the requirements for reviews that are traceable to the regulation itself:

- *Documented*: No more silent reviews in your head. The results of the review, the action items, and resolution of the action items should be documented for the DHF. Documentation should include what was reviewed, when it was reviewed, the results of the review and who participated in the review.
- *Planned*: Reviews should be planned events at control points in the project life cycle where the results of the review can be optimally acted upon.
- *Appropriate stages*: Note that this is plural, which implies that a one-time *impromptu* review in the hallway is not what the regulators had in mind. Reviews throughout the life cycle are more likely what was intended.
- *Functional representatives and specialists*: Various stakeholders who might have an opinion about how the details of the design evolve, or about how the design satisfies the user needs (design inputs), or whether the requirements and implementation are testable should be part of the reviews. Reviews that meet regulatory muster are generally not just between officemates or disinterested parties who will expedite the completion.
- *Individual(s) without direct responsibility*: Sometimes this is referred to as an independent reviewer. One cannot adequately review one's own work. The mind too readily fills in the gaps. Ambiguities are never resolved because they are generally clear to the person who created them. So one is left only with documents and designs that are only clear to the originator himself or herself.

## Why the Focus on Reviews?

The regulatory requirements for review are numerous. In Chapter 4, we looked at the regulatory requirements for reviews and approvals and discussed the role of the manufacturer's organization in quality in general and validation specifically. Clearly, reviews are considered by regulators to be important, but in my personal

experience, few organizations take them as seriously as they should. Few organizations have done anything to create or improve a review process.

So, what is so great about reviews? Perhaps by understanding what reviews could accomplish we will be lead to a better understanding of why they are so highly recommended.

In the context of verification and validation, a review can usually be considered to be a verification activity. When a design output is reviewed one of the primary objectives is to assure that the design output accurately has expanded upon the design inputs. As shown in Figure 9.7, at each phase of the life cycle, a review (represented by the dashed lines) verifies that the design intent expressed in the design inputs is carried through to the design outputs.

What does that mean, exactly? 820.30(c) states that requirements reviews should assure that they are complete, unambiguous, and nonconflicting. The GPSV offers further guidance (Section 5.2.2) by recommending that requirements should be reviewed for “accuracy, completeness, consistency, testability, correctness, and clarity.”

Accuracy, correctness, and completeness relate directly to the goal of carrying design input intent through to the design outputs. Traceability is a tool that facilitates that analysis and will be described later in this chapter. Although these attributes of reviews are embedded in regulation related to requirements review, the same should apply to later phase reviews. Designs should be accurate, correct, and complete representations of the requirements. The software itself should be an accurate, correct, and complete representation of the design. Reviewing for accuracy, correctness, and completeness helps to keep the design process on track by keeping it aligned with the product level design inputs (user, clinician, patient needs, etc.).

What about those other attributes: unambiguous (clear), nonconflicting, consistent, and testable? These are quality attributes of the design outputs. In addition to reviewing for the accurate promulgation of design intent, the design outputs must be usable as design inputs to succeeding phases of the life cycle. To be *usable* they need to be clear, nonconflicting, consistent, and testable.

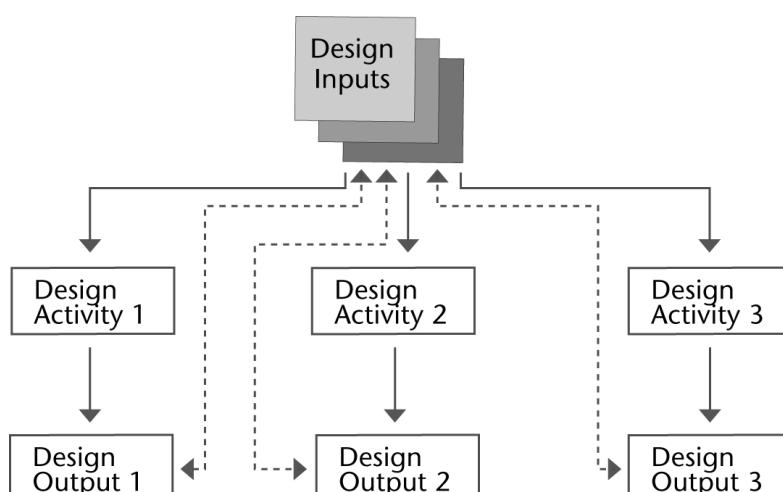


Figure 9.7 The role of review in the design control process.

Managing a development project is much like navigating a ship from an origin (product needs) to a destination (finished product). Navigation is based on a predictor-corrector process. Location is predicted (dead reckoning) based on last known location, heading, speed, and time. In a perfect world, if the dead reckoning calculations are correct, one's position would be known exactly, and the destination would appear at the bow of the ship every time. Unfortunately, other factors come into play such as wind and currents that cause the ship to drift from the intended course and the actual position to be significantly different from what the dead reckoning calculations would predict. It is possible, even likely, that one would arrive at the wrong destination despite perfectly accurate speed and heading measurements, and perfectly correct calculations. For this reason, navigators periodically correct their position (a fix) on a chart based on land observations, celestial measurements, or other instrumented means. Getting an occasional fix to correct the course keeps the ship from arriving at the wrong destination or from running aground as it deviates from its predicted course.

Design and development activities also can be executed perfectly and still develop a product that does not fill the needs of the intended users. In this case ambiguous, conflicting, inconsistent, or untestable design outputs can cause the equivalent of drift in the direction of development just as wind and currents cause drift in the course of a ship. Design reviews therefore must achieve two things. First, they must assure that the “dead reckoning calculations” are accurate, correct, and complete. Second, the reviews must “get a fix” in position in the project by assessing the accuracy, correctness, and completeness of the design outputs and correcting for any deficiencies from the intended course toward the intended destination of the product design and its ability to meet the intended use for its intended users.

It should be obvious what is meant by designing the wrong device. Several example stories were presented earlier in the book. What is the equivalent of “running aground” as the project goes off course? Missing or ignoring certain product level requirements until late in the life cycle can lead to major redesigns. The later this occurs, the more damaging and costly it is to repair. Frequently, the changes are too costly and ignored ... leading to producing the “wrong device”. Issues identified early in the life cycle that are ignored until late in the project can have the same effect.

In a postmortem analysis of a multiyear project our company did for a client several years ago, we were able to use metrics from engineers' timesheets, and metrics from our defect tracking system to look at some project statistics. One of the most surprising results that came out of the study was that the cost of finding a defect through system level testing in this project was 95 times more expensive than the cost of finding a defect in a review. That did not include the cost to correct the defect which almost assuredly was at least 95 times more expensive in later phases of the life cycle. That sounds like a huge success, right? Unfortunately the client shut down the review activity because it was finding too many defects (see “Be Ready to Deal with What You Find” below). Doubly unfortunate was the fact that some of the same defects seen in reviews had to be re-identified during system level testing, costing the client almost 100 times what it would have had the review process been followed to completion.

## What Is Meant by a Review?

We will not belabor this topic with a long list of things to be reviewed. It would differ from project to project anyway. Let us start by saying that all design outputs and project plans should be reviewed. Product level design inputs should be reviewed for all the same attributes described above. The product level design inputs should be challenged during the review process. Often something that is presented as a requirement turns out not to be a requirement when challenged, or when the originator of the requirement is informed of the potential cost, complexity, or risk involved in implementing the requirement.

Rather than focusing on the subject of the review, let us discuss various types of reviews. This is worthy to note, and worthy of distinguishing what type of a review is required in a development or validation plan. Below, these are ranked in order of least formal to most formal:

*Peer reviews.* These are often informal, *impromptu*, and not documented well, if at all. However they should be encouraged because two heads are better than one, right? They should be encouraged but should not substitute for the more formal reviews below. These should be limited to reviewing an interpretation of a requirement, preliminarily reviewing a design construct for validity and impact on other areas of the design, preliminarily reviewing implementation constructs for potential other use of language, or programming style. Participants are limited to a peer, manager, or other trusted consultant.

*Technical evaluations.* Often these are more formal and comprehensive versions of peer reviews. Participants include other competent technical reviewers from the development and validation teams. Traceability is reviewed as a mechanism of assessing accuracy, correctness, and completeness of the work being reviewed. The design output being reviewed is also assessed for ambiguity, conflict with other parts of the design, consistency, and testability. These technical evaluations tend to be very detailed, and therefore the reviewer list should be chosen (and limited) according to who will be competent (or interested) in participating in that level of detailed review.

*Design reviews.* These are more “big picture” reviews that are intended to include a broader list of participants. Participants in design reviews are not only from the technical developer and tester group, but also from clinical, user, legal, regulatory, and quality stakeholder groups. The purpose of these reviews is to assess whether the project is still on course to produce the intended device that will be both effective and safe.

*Management reviews.* These can be summaries of the other reviews. Management needs to know that the more technical levels of review took place. Management needs adequate documentation of the results of those reviews and the resolutions of any issues raised. Why is this necessary? Management eventually takes on the legal responsibility for the device that is implemented. That does not mean that executive management needs to attend each and every review at all levels. However, they do need credible assurance that the reviews took place, were treated

with an appropriate level of professionalism, were adequately documented, and that any issues that were raised in the reviews were appropriately resolved. Management needs to know what risks (business, regulatory, and safety risks) the company and the manager personally are taking in moving forward in the development/validation project.

The IEEE Standard for Software Reviews [2] offers a slightly different breakdown in types of reviews: management reviews, technical reviews, inspections, walk-throughs, and audits. There are adequate reasons in certain situations to differentiate at least two different levels. There are entire books available on individual types of reviews. Going any deeper is beyond the scope of this text. However, IEEE standard is good for breaking down what types of design outputs should be subjected to what types of reviews, and recommends lists of participants for each review.

### **Who Should Be Participating in the Reviews?**

This topic also will not be belabored with detailed lists that would differ from company to company and device to device. The above discussion on types of reviews mentions how the participants may change for various types of reviews. At some point in the development/validation life cycle customers, clinicians, users, designers, developers, testers, quality engineers, marketing experts, executive management, and regulatory compliance experts should all be involved in reviews.

It is important to recognize that all different reviewer stakeholders will be, and should be, looking for different aspects of the item being reviewed. It is best to be specific about this and inform each reviewer of what is expected of them in their participation. Doing so will free the reviewers to focus on their area of expertise, and allow them pay less attention to topics outside their area of expertise. This is more efficient for everyone, and raises the likelihood that reviewers will find time for a much less daunting task. The RASCI charts that were discussed in Chapter 4 are an excellent way of crystallizing thought and documenting expectations for review responsibilities.

One of the side effects of including a diverse group of reviewers is that the medium for communication of the topic being reviewed may not be appropriate for the entire list of reviewers. For example, the software requirements for the user interface for a device may benefit from the review of intended users, marketing experts, and the software developers and testers. The developers and testers will be interested in the details of each individual requirement. Requirements documents of hundreds or even thousands of pages in length are common. Would anyone expect an intended user to read that? If they did, would anyone expect any meaningful comments back?

Less technical reviewers may need another medium to review. Flow charts or rapid prototypes of the user interface may be more appropriate for collecting meaningful input from nontechnical reviewers. However, these are not adequate substitutes for the technical developer and tester reviewers. Unfortunately, this adds some overhead in creating different views of the topic to be reviewed, but the quality of participation from the reviewers should give more than an adequate return on the

investment. In situations like this, it may pay to break the review into several sessions to review increasingly technical aspects of the topic with a smaller reviewer panel.

### How Reviews Are Conducted

As mentioned above, there are entire books devoted to how to conduct a single type of review. Comments here will be limited to some valuable general suggestions that experience has shown to make the review process more valuable.

#### Work with the Reviewers Before the Review

Do not assume that your reviewers know how to review a specific design output, or what is expected of them. They will need some training on the review process in general. They will also need a clear understanding of their specific role in the review process. Be specific and document the role expectations for all reviewers to see. As mentioned previously, the RASCI charts are a good way to communicate the roles and responsibilities for each review.

Once the reviewers know what is expected of them, they will need some coaching on how to perform their assigned roles. It is a good idea to provide them with suggestions for what types of problems they are responsible for finding, and, if possible, examples of what those types of problems might look like. For some types of reviews such as code inspections, we have found it useful to use checklists of common errors for the engineers to use as a guide. As with everything, checklists have their advantages and disadvantages. On the positive side, a checklist does help provide a more consistent result from module to module and reviewer to reviewer. If it is a good checklist, the result is good. On the negative, reviewers with checklists tend to focus on the items on the checklist and turn off the critical thinking needed to find defects that were not anticipated by the checklist.

#### Not All Defect Types Share the Same Degree of Difficulty

Some review tasks are simple, and others are very difficult. The scale shown in Figure 9.8 arranges review findings on a relative difficulty scale. Reviews that only turn up spelling and grammar/syntax errors are probably of limited usefulness, yet

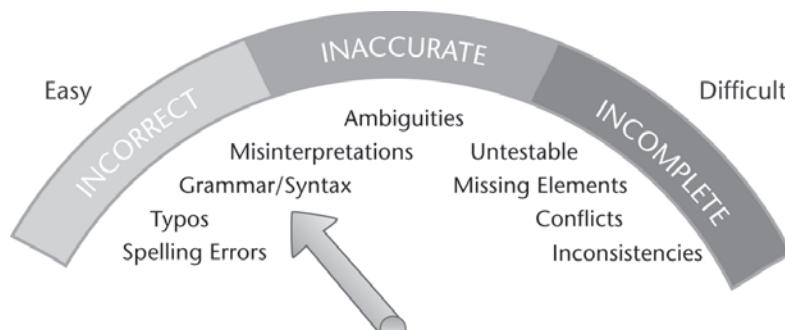


Figure 9.8 Difficulty scale for review findings.

it is common for over 80% of the review findings to be in this category. A technique we have added to our own review procedures is to break the review into three passes. Spelling/typo/grammar/syntax findings are not permitted on the first and second pass reviews (which are devoted to overall structure and detailed content). This forces the reviewers to think about the more difficult to find issues like missing/conflicting/inconsistent design output elements. There is little value correcting spelling on something that might change simply because it is wrong. Additionally, it is easy for a reviewer to feel very productive if dozens of simple typographical problems are logged. That can lull a reviewer into a false sense of thoroughness, allowing more important defects to escape detection.

What makes some defects easy to find and others nearly impossible? Much of it has to do with the scope of examination and understanding required to find a defect. Obviously, to find spelling or grammar/syntax errors one only needs a command of the natural or computer language being reviewed. The scope of the defect is limited to a word or sentence in natural language, or a variable/constant name or line of code in computer source language. These kinds of defects are so easy to find that software spell/grammar/syntax checkers are good at finding most of them. At the other end of the scale, conflicting or inconsistent design output elements require a more in-depth awareness, understanding, and memory of the design elements that could be separated by hundreds of pages of text or thousands of lines of code. These kinds of findings are usually only found by reviewers who are very experienced at reviewing, and very experienced with the device product line and/or have a broad understanding of the design outputs of the project.

### Know When to Start a Review and Know When to End It

Two concepts introduced by the IEEE Standard for Software Reviews are the entry and exit criteria for reviews. Both are worth documenting in the development and validation plans that should detail what reviews are to be held. The entry criteria obviously will include the completion of the review inputs. Additionally, the stability of the input(s) should be a consideration for starting a review. There is little value expending significant resources reviewing a design output that is still changing hourly.

Likewise, predetermined exit criteria will help the review team understand their objectives for completing a review. Stating the exit criteria also adds an increased sense of importance of the review documentation that will show the exit criteria have been met. Typically the exit criteria specify that any action items have been implemented and verified to the satisfaction of the review team. Alternatively, if the review findings are not to be implemented immediately, some means for capturing the findings and assuring their proper disposition at some later time could be a perfectly acceptable (and practical) exit criterion. Exit criteria may also specify acceptable exceptions for exiting a review.

### Spend Some Time Defining a Review Process and Continually Improve It

This goes hand-in-hand with the earlier observation that one cannot expect all reviewers to know how to review. The more review expectations that can be docu-

mented, the more likely that the reviewers will reach the desired level of performance.

Review procedures are defined in many textbooks, some of which are listed in the references at the end of this chapter. One process that we have found to be effective is the Fagan Inspection Process created by Michael Fagan (<http://www.mfagan.com>). This is a very rigidly structured and detailed process primarily intended for code inspections, but which also works equally well for requirements and design reviews. Only three elements of the Fagan process will be mentioned here. In my opinion, these are among the most productive of the elements of the Fagan process. (Admittedly, some of these have been modified for my own purposes. My apologies to Mr. Fagan if he doesn't recognize them.)

### *Preparation*

There is little reason to waste a reviewer's time in a review meeting if all reviewers are not adequately prepared. Recall that the reviewer list is designed to cover all viewpoints of the design output being reviewed. If even one person is not prepared, that viewpoint will not be adequately represented.

Reviewers should be told how they should prepare (each may be different since they are looking for different things). At the beginning of the review, reviewers are asked how much time was spent preparing for the review. These times are logged into the review minutes. If it is determined that the preparation is inadequate for a valuable review, the review is rescheduled.

### *Limit the Length of Each Review Session*

Reviewers are, after all, only human. If the expectation is for a value-adding review experience, it will likely be intense. It is difficult for most humans to focus at that level for more than 90 minutes without suffering some decrease in performance level.

Look at the review notes for your last review. How many findings were logged for the first section of the item being reviewed, and how many were logged for the last section? It's not uncommon for there to be five to ten times more findings in early sections than in the later sections. Is that because the last sections of every document are always five to ten times better than the opening sections? Probably not. More likely this is documented evidence of the loss of focus of the reviewers. Limit review sessions to 90 minutes. Most reviews cannot be completed in a single session. Limiting the length of a session not only assures that reviewers will be fresh, but the limited time of the session also adds a sense of urgency to make progress in the limited time that does not always seem to be present in open-ended sessions.

### *The Role of the Reader*

I will admit that I probably rolled my eyes when I first heard this technique, but in my opinion, it is the single best technique I have ever seen for finding ambiguities in requirements, as well as in source code.

The technique is simple. At the review meeting an individual is designated as reader (actually, the designation is made before the meeting so the reader can properly prepare). The reader cannot be the individual responsible for creating the material being reviewed. (Being reader can be tiring, so there's no reason it can't be a

rotating assignment.) The reader goes through the item being reviewed object by object. An “object” in this context means a requirement, design element, sentence/paragraph, line of code, and so forth. The reader “reads” each object in his or her own words. Really this is more interpreting than reading, but it is very important that the reader read the object silently to himself or herself, then verbalize the meaning in his or her own words.

If all the reviewers and the author of the material being reviewed agree with the reader’s interpretation, then the object was not ambiguous. However, it is amazing how high a percentage of objects in a given review are misinterpreted.

Of course, the review also looks for the other attributes of correctness, accuracy, completeness, and so forth after the ambiguity question is answered. Often review comments made during preparation are not necessary in the review meeting once the ambiguity is resolved.

### Start with the Right Attitude about Reviews

Getting good results from reviews and from verification and validation testing has a lot to do with the attitude with which you pursue the activity. Our attitudes often act as self-fulfilling prophecies. If one grudgingly undertakes a review feeling it is a waste of time, chances are it will be a waste of time. On the other hand, if one takes on the review task knowing there are defects in the material being reviewed, and feeling a sense of accomplishment when they are found, the chances that the defects will be found is markedly improved.

Attitude at review meetings is important for the review group. The objective of reviews and review meetings is simply to find defects. It is not an objective, or appropriate, to criticize individuals for misunderstanding or making mistakes. It is not an appropriate opportunity to make oneself look smarter at someone else’s expense. It is important to keep one’s mind focused on reviewing the material to improve the product. Leave pride, politics, and arrogance for another time (if ever). The creators of the items being reviewed also need to adopt appropriate attitudes for reviews. There is no need to be defensive about review findings. Simply log them, understand what should be done about them, and move on to the next. Each item found at a review is a success because it saved the organization time and money by discovering it early in the life cycle.

### Be Ready to Deal with What You Find

The organization itself also needs an appropriate attitude about review findings and test results. The last thing an organization should consider is avoiding reviews because they find too many defects. The next-to-last thing an organization should consider is obscuring the volume of findings by combining them, closing them without proper consideration, or simply deleting them. I have encountered all these behaviors in my experience and probably will again. True, one may have tried to review (or test) a design output before it was ready. That calls for postponement of the review, and eventual, thoughtful, verification and closure of defects that were found before the review postponement, not deletion of the “evidence” and cancellation of further reviews!

Unfortunately, organizations seem to have an inability to deal with unpleasant facts or bad news. That is behind some of the behavior that resists thorough reviews, or careful record-keeping for defect management. The same is true of scheduling, budgeting, or any other tool that is capable of delivering bad news. The organizational reflex seems to be to bury the unpleasant results, discredit the tool or methodology that created the undesired facts, and proceed blindly with fewer facts to get in the way. For some reason it seems to be preferable to ignore inconvenient facts (when something might have been done about it) only to be “surprised” by the results later (when it is much more costly to address them).

How does an organization fix that? It is really the same attitude adjustment that is needed at the personal level. An attitude needs to be fostered that recognizes success in the prolific identification of defects, and the careful, thoughtful management of them through resolution and closure. Yes, sometimes it takes a strong stomach to do this, but anything less is only postponing bad news for a later day when it is bound to be worse.

## Traceability

Traceability is a tool that often is grudgingly implemented, and, if implemented is often not leveraged to take advantage of the information it can provide to a development or validation team. Those who master traceability recognize its value and often overtrace. Commercial software tools have evolved to eliminate much of the labor involved in maintaining traces. Some tools even identify “broken traces” (i.e., traces that need to be reexamined because either the source or destination of the trace has changed since the trace link was established).

### Why Traceability?

Before delving into the regulatory opinions about traceability, let us examine why traceability is a good engineering practice regardless of regulatory opinion.

#### Origins of Requirements

Some requirements are less flexible than others. For example, requirements that exist to meet a requirement in an industry standard generally should not be altered. Requirements that relate to a risk control measure cannot be altered or deleted unless an alternative solution achieves the same level of risk control. Knowing the origin (ancestry) of a requirement, design element, or even a test is valuable information for managing a project.

#### Metrics

Traces that are assigned as work is completed provide good completion metrics for project management. For example, “There are 1,500 software requirements, and we’ve defined design elements for 1,000 of them. Our design effort is about 67% complete.”

## Analysis

Accurate traces that are kept up to date as the project evolves provide a wealth of information for detecting the quality of the requirements and design descriptions. Inadequacies in either will show up as orphaned elements downstream. For example, a design element without a trace link to a requirement implies a missing requirement (or an extra design element that is not required). Specific examples will be noted below with specific types of traces.

## Safety Tracing

Anything related to safety is a primary goal of traceability. Traceability is used to assure that safety related requirements are created from risk analysis activities, and that those requirements propagate down into designs, implementations, and tests that verify their ability to control the risk they were designed to control.

## Regulatory Background

The QSRs do not specifically mention a requirement for traceability, but certainly the FDA's GPSV recognizes the value of traceability, and mentions it for numerous applications:

### Section 3.1.2

... validation of software typically includes evidence that all software requirements have been implemented correctly and completely and are traceable to system requirements.

### Section 5.2.2

A software requirements traceability analysis should be conducted to trace software requirements to (and from) system requirements and to risk analysis results.

### Section 5.2.3

A traceability analysis should be conducted to verify that the software design implements all of the software requirements. As a technique for identifying where requirements are not sufficient, the traceability analysis should also verify that all aspects of the design are traceable to software requirements.

### Section 5.2.4

A source code traceability analysis is an important tool to verify that all code is linked to established specifications and established test procedures. A source code traceability analysis should be conducted and documented to verify that:

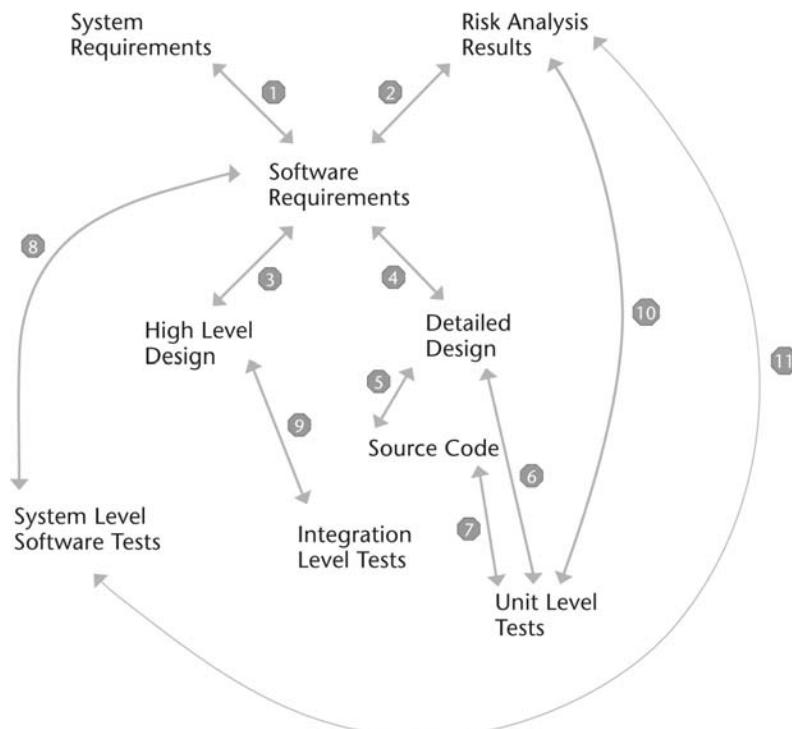
- Each element of the software design specification has been implemented in code;
- Modules and functions implemented in code can be traced back to an element in the software design specification and to the risk analysis;
- Tests for modules and functions can be traced back to an element in the software design specification and to the risk analysis; and
- Tests for modules and functions can be traced to source code for the same modules and functions.

### Section 5.2.5

Control measures (e.g., a traceability analysis) should be used to ensure that the intended (test) coverage is achieved ...

- Traceability Analysis— Testing
- Unit (Module) Tests to Detailed Design
- Integration Tests to High Level Design
- System Tests to Software Requirements....

For those of you counting, the guidance quotations on traceability dissected above from the GPSV mention 11 separate trace links that are summarized in Figure 9.9. One might debate whether all of the links are necessary, or whether this is the right set of trace links, or even whether these are enough links. For now let us just accept that these are the links recommended in the GPSV, and examine them to



**Figure 9.9** Trace links recommended by the GPSV.

understand why the regulators may have felt that these specific trace links may be of importance.

1. *System requirements to/from software requirements:* Traces from system requirements are needed for assurance that all system requirements are accounted for in software and other lower-level requirements. Traces from software to system requirements are needed for future inquiries on the source of a software requirement. For example, one might ask if a software requirement exists because it is a system requirement. If not, there may be some more latitude to change the software requirement. All system requirements should trace to something. Software requirements may not necessarily trace back to system requirements.
2. *Risk analysis (control measures) to software requirements:* Traces from risk control measures are needed for assurance that all risk control requirements are accounted for in software and other lower-level requirements. Traces from software to risk control measures are needed for future inquiries on the source of a software requirement. For example, one might ask if a software requirement exists because it is a risk control measure. All risk control measures should trace to something. Not all software requirements will trace back to risk analysis or risk control measures.
3. *Software requirements to high-level design and detailed design:* In theory, one would want to see each software requirement trace to some design element in either the high level or detail designs, or both. Conversely, each design element should trace back to some software requirement. If significant numbers of “orphan” design elements exist, it may be an indication that the requirements do not have an adequate amount detail to steer the design. It could also be an indicator of “gold plating;” that is, designers adding functionality that was not anticipated or needed by the authors of the requirements. Admittedly, this trace is sometimes difficult to implement. It is quite possible that one requirement could trace to multiple design elements (one to many). The converse (many to one) is also a possibility. Since either of these possibilities makes it difficult to analyze and maintain the trace, one might take this into account early in the design phase, and plan the organization of the design documents to closely match the organization of the requirements. It takes some creativity to make this trace manageable, but even the existence of a trace link (assuming it is accurate) provides assurance that all requirements are designed, and that there is no extra design that is not required.
4. *Detailed design to source code:* This trace is useful for finding what software units are involved in the implementation of a detailed design element. Design elements without traces to source code are not implemented. Source code without a link to a detailed design element exists for unknown reasons. This implies that the detailed design was not detailed enough (and will be difficult to maintain, and will not be tested thoroughly), or that gold plating is taking place at the code level. Code without a design trace will likely escape adequate testing, since unit level testing tests code against the detailed design description.

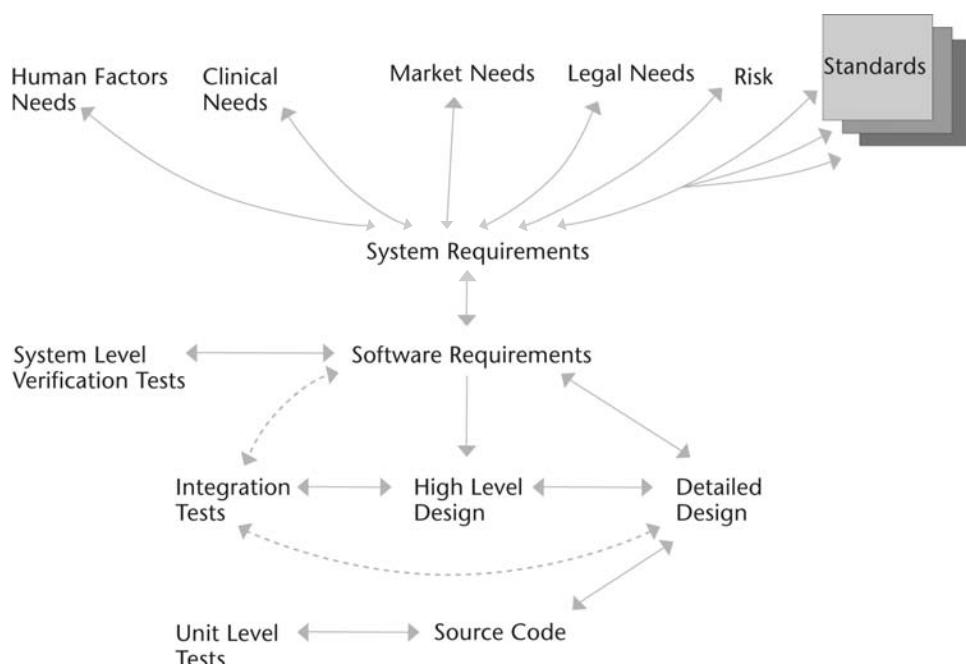
A simple means of organizing and making the trace understandable is to tie the name of the source code unit to a corresponding section name in the detailed design. For example, a detailed design element named round\_up that is implemented by a source code function named round\_up really requires no other explanation. Implicit naming is not sufficient to achieve the purposes for these traces, however. Two other pieces of information are needed. First, the source code function should be linked to the source file in which it resides (simply so one knows where to find it). That can be detailed in the detailed design, or can be a simple source file contents listing. Second, some trace must exist from detailed design to source file contents. This is needed for an analysis that shows each design element traces to a software unit and vice versa.

5. *Detailed design to unit level tests:* The main value of this trace is assurance that each detailed design element is covered by a unit level test (or rationale for why there is no test). Assuming that all software is represented by design descriptions, this traces assumes that all software will be tested at the unit level.
6. *Source code to unit level tests:* If a one-to-one association of detailed design element to source code unit exists, this trace is redundant (i.e., if there is a unit test for a design element, then it exists for the code that implements the design element). If the mapping from design element to source code is not one-to-one, this trace will be much more challenging, as will the design of the unit test itself. One-to-one mapping between detailed design elements and source code units is a highly recommended practice that makes traceability and testing much simpler to design, document, and explain.
7. *Software requirements to system level software tests:* One need for this is obvious; a trace from software requirement to system level software test assures that each requirement has been tested. The reverse trace can be used to find tests that exist that do not trace back to a requirement. This may be a sign that the requirements do not have adequate detail, or that there is an attempt (possibly out of desperation on the part of the tester) to detail the requirements and/or design in the tests themselves.
8. *High level design to integration level tests:* If we accept that integration level tests exist primarily to test the interfaces between major elements of the software design, then those interfaces are most easily designed and documented in a high level design description. High level designs are often graphic in nature (block diagrams, flow diagrams, etc.). That can make traceability a challenge, but since this is a relatively low-resolution trace, text accompanying the graphic representations of the high level design can easily be used for traceability without adding an undue burden.
9. *Unit level tests to risk analysis results and system level software tests to risk analysis:* These traces, it could be argued, are also redundant with other traces. For example, if system level software tests trace back to software requirements, which trace to risk analysis results, then by inference, the system level software tests also trace (indirectly) back to the risk analysis results. A similar argument could be made for the unit level tests.

### Traceability Beyond the Regulatory Guidance

The above are the trace links that are mentioned in the GPSV. That does not mean they are the only ones possible, or that they are the only useful ones. For example, several of the links described above cite the value of tracing requirements to their origin. However, the trace back stops at system requirements. Where do system requirements come from? System requirements originate in user needs, market needs, clinical needs, and legal and regulatory needs. Additionally, some early research may have been conducted to determine human factors requirements and recommendations. Finally, requirements may have originated from industry standards (such as the ISO 60601 safety standards) or even regulatory guidance documents that are general, or specific to a device type (such as the AAMI ID26:2004 standard on infusion pumps). Tracing system requirements to their specific origins is very valuable for all the same reasons that we would want to trace software requirements to system requirements. So how might this change the trace schema of Figure 9.9? There are a number of ways to organize design outputs. There is no single right way. There are trade-offs for any method used, so let us discuss what factors might influence one's choice.

Figure 9.10 shows a purist's top-down traceability schema. All of the identified needs are represented in the system requirements. From there the trace flow is similar to that as shown in Figure 9.9. This is a predictable top-down organization that works well for many projects. Note that this figure has introduced traces to some number of standards. If something is required by a standard, it makes sense that it would be a system requirement, right? The answer is, yes, but there are a number of practical situations in which this breaks down.



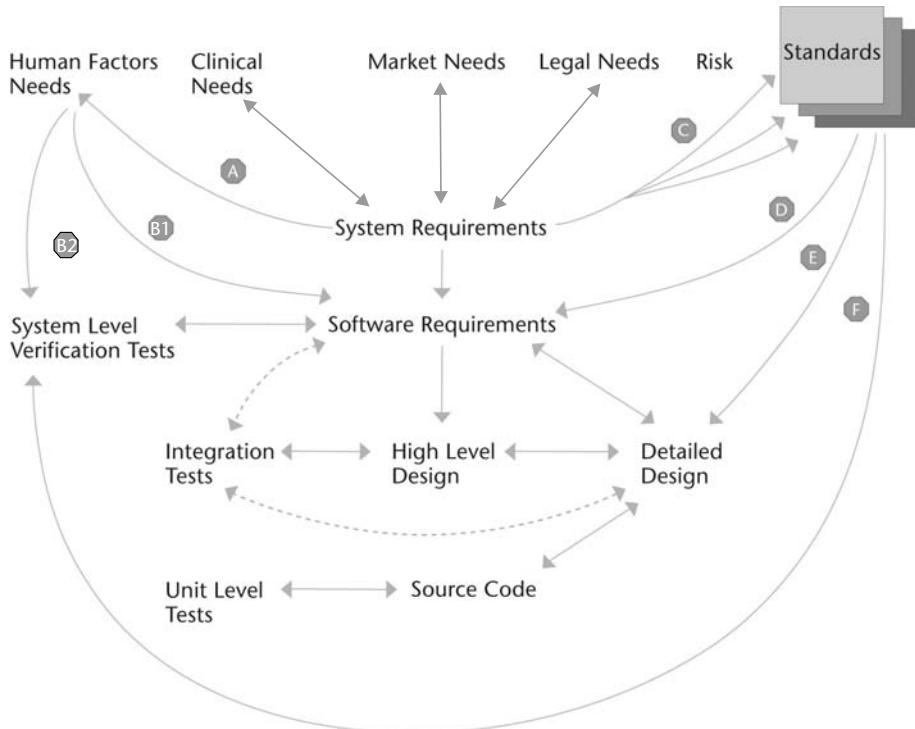
**Figure 9.10** Pure top-down organization.

For industry standards specifically, this could result in many hundreds of requirements that would be repeated in the system requirements. (We recently ran into a project to develop a device that had over 30 applicable industry standards that applied to the specific type of device.) Many of the device-specific standards contain specific requirements for how the performance of the device is to be tested and measured (to support standardized reporting of specifications that facilitates comparisons of devices). This is objectionable even to the purists since it will result in test requirements that are embedded as system requirements. The “pure top-down” structure would force those requirements down through software requirements, and so forth. The pure implementation certainly has broken down at the point that “requirements” end up in unexpected locations, or simply get repeated verbatim to satisfy the trace structure—especially since one of the advantages of the pure model was its predictability. For projects that have a large number of needs inputs, or must comply with a number of detailed standards, this structure will be problematic and will result in systems requirements with a burdensome amount of detail.

An alternative trace schema that does work better for top-heavy requirements is shown in Figure 9.11. This schema covers the same inputs and outputs; only the trace links have changed. Several of the links are labeled for reference in the diagram. Links A and B refer to the links to and from the human factors needs. Note that both of these links are unidirectional. The link from the system requirements to the human factors needs (which itself may be voluminous) might be a single requirement in the system requirements, reading something like, “The design of the device shall incorporate all human factors requirements as described in the Human Factors Needs Analysis—Version 1.0.” The B links from human factors needs to software requirements and system level tests represent identifiable requirements (B1), and unquantifiable design goals that are to be tested that may not explicitly be represented by requirements (B2). What is meant by unquantifiable design goals? Suppose the human factors needs analysis contains a statement like, “Clinicians must be able to respond to an alarm appropriately in less than 3 seconds.” That statement (requirement) may result in a number of specific software requirements to attempt to meet that goal (like menu depth, software response time, display clutter, etc.) that are all verifiable in the normal software verification test path. (We discuss whether this is a good requirement or not in a later chapter.) However, one really doesn’t know if the 3-second goal is met until it is tested in a clinical environment with intended users. Therefore it seems appropriate that the human factors needs requirement is directly testable, and thus deserving of its own trace link. Note that directly testing the human factors needs is a form of validation testing.

Traceability to standards is somewhat similar. Just to clarify, the standards referenced in Figure 9.11 are device specific standards that specify device performance, design specifications, safety, or testing standards. These are not quality system standards like the 62304 life cycle processes standard, or the 14971 the risk management standard. Quality system standards probably are best traceable to the manufacturer’s quality system (to assure compliance), but not to specific design outputs.

Trace links C are for system requirements for conformance to device-specific standards. As shown in Figure 9.11 these are unidirectional for our top-heavy



**Figure 9.11** Alternative organization for top-heavy requirements.

schema. The links in D, E, and F are one-way links from the device specific standards to whichever design output would best record and expand upon the specific requirement in the standard. As the figure shows, in some cases the best place to satisfy the standard requirement may be in the software (or other) requirements, while other requirements may best be satisfied at the design level. Standardized test requirements obviously are best treated in some level of test documentation.

Take for example the following requirements from an industry device specific standard on infusion pumps (AAMI ID26:2004):

- ... a) the audible alarm shall be able to produce a sound-pressure level (or, if adjustable, a maximum level) of at least 65 dB(A) at 1 m, and shall not be by the OPERATOR externally adjustable below 45 dB(A) at 1 m;
- b) the audible alarm silence period of the EQUIPMENT in operation shall not exceed 2 min;
- c) the visual alarm shall continue to operate during the audible alarm silence period;
- d) means shall be provided to enable the OPERATOR to check the operation of audible and visual alarms.

Compliance is checked by measuring the A-weighted sound pressure level, with an instrument complying with the requirements for a type 1 instrument laid down in IEC 60651 or IEC 60804, as follows:

The pump and the microphone are placed in free-field conditions (according to ISO 3744), at a height of 1.5 m from the reflecting plane. The distance between the pump and the microphone shall be 1 m. The background noise level shall be at least 10 dB(A) below the sound pressure level to be measured. During the test, microphone orientation should be toward, but in the lowest horizontal sound power direction from, the pump [3].

Requirement a) is a compound requirement that would partially be satisfied by hardware (electrical) requirements and partially satisfied by software requirements (if the volume adjustment is software controlled). Requirements b), c), and d) are probably best satisfied in software requirements. Further, it is easy to see how each of these requirements in the standard not only would trace to the software requirements, but also would be expanded upon in the software requirements to detail how the device will specifically meet these standard requirements.

After the itemized requirements above are two paragraphs of additional requirements for how the sound volume is to be measured for compliance to the standard. This is a *test* requirement and following our top-heavy schema these should trace directly to some high level test procedure.

It should be noted that the last two paragraphs of the quoted section contain three references to other applicable standards (60651, 60804, and 3744). Following the chain of references for applicable requirements makes the trace schema even more top-heavy. This is exactly how our recent project ended up tracing to over 30 individual industry standards.

To reiterate the point being made with these different trace structures, either of the two examples presented can be made to work. The pure top-down schema will likely result in system requirements that are very detailed (and voluminous). From a practicality perspective, the system requirements should serve the broadest audience of any of the design outputs. It makes sense to keep it as terse as possible to keep it readable. All that is really required in the system requirements is a single requirement, such as, “The device design shall comply with applicable requirements of the AAMI ID26:2004 standard.”

The details of the applicable requirements will clutter the high level system requirements, making it difficult for the broad audience to review. Furthermore, the requirements from standards require little review anyway. They are requirements if the standard is to be followed. No further discussion is warranted.

### **Practical Considerations: How It Is Done**

If you have never been responsible for traceability in a project, you may have understood the above discussion and diagrams on trace organization, but may be wondering just how this gets done and what it looks like in a design history file.

### **Trace Tools**

There are a number of ways traces can be documented. Some trace a section of a child-level document to a section of a parent-level document (e.g., system requirements to software requirements) simply by numbering the sections of the parent and child documents identically. This is inferred traceability. It is low-resolution

traceability, and is problematic if any child-level objects are shared among parent-level requirements. Low-resolution traces are less useful for analysis and metrics purposes, but do meet the need for traceability as suggested in the regulatory guidance. (It may be compliant, but if it's not useful, why would you choose this approach?)

Others tag the objects of a child document with the section number of the parent that had the requirement fulfilled by the child document object. Let us call this “parent section number” tracing just to give it a name. Using the example above from AAMI ID26:2004, an infusion pump’s system requirements may contain an object such as, “{AAMI-26:51.109} The volume control feature of the software shall limit adjustments from the maximum volume of the pump down to a value no less than 45 dB(A) at 1 m.”

The reference inside the braces is the trace back to the AAMI ID26:2004 parent-level document, the 51.109 is the section number of the standard that is partially fulfilled by this requirement. This is somewhat better than the inferred method. It is high-resolution at the child end of the trace, but the resolution of the parent document section is low resolution. Why is that a problem? The Section 51.109 quotation above has between five and ten requirements depending on how you count them. Consequently, the trace from the system requirements example above could trace to any one (or more) of the five to ten requirements of Section 51.109. To make matters worse, this system requirement may only partially fulfill one or more of the requirements. The point is, low-resolution traces require a lot of interpretation, interpolation, and mind-reading of the reviewers and those charged with implementing the next level of the design. This method of tracing is unidirectional. We can tell what parent-level requirement was the genesis of the child object while looking at the child document. However, we cannot look at the parent and tell where the child document(s) have implemented the requirement without a lot of manual file opening and searching of child documents. Again, this kind of trace may meet regulatory oversight, but is only marginally useful.

Another method related to the above two is to use a spreadsheet program or a word processor’s table features to handle the “trace matrix.” The matrix can be implemented in both directions. Some automation may even be possible with macros to automatically create the reverse links from the forward links (or vice versa). The resolution of the requirement or object naming (i.e., section numbers, tags, labels) will limit the value of this method. For instance, if the matrix traces section numbers in the parent to section numbers in the child, it is of no more value than the inferred method of tracing mentioned above.

Thankfully, there are software tools available for requirements management that facilitate and automate traceability needs. These tools tend to be somewhat expensive, but in my opinion they more than pay for themselves in productivity and make possible the high-resolution traces that make the whole traceability exercise a value-adding activity.

Requirements management tools assist with trace creation, maintenance, and analysis in several major ways:

*Unique numbering of requirements.* It probably doesn’t seem like much of a feature, but having a unique label for an objects that doesn’t change when new objects

are inserted or old objects are deleted is absolutely essential for maintaining a trace. Imagine how much work would be required in the worksheet method of tracing if a new requirement were inserted before 51.109 of our example, making it 51.110. Every trace in section 51 after 109 would have to be renumbered. The requirements management tools create a separate unique label that is separate from any section numbering or other material content of the document. The labels don't change as the document is edited. Further, the tool keeps a record of the last requirement label issued, and automatically generates a new unique label when the next requirement is created, regardless of its position in the document.

*Trace creation.* Requirements management tools often (if not always) provide a means to facilitate creation of the traces. Don't get too excited yet. The tool doesn't do the thinking for you. Most tools provide a drag and drop capability for creating a trace link from one document to another. The analyst points at a child object, clicks and holds the mouse button, and drags the cursor to the requirement in the parent document. Multiple links are supported in either direction. This is a very natural way of creating links that eliminates the need for transcription of identifying labels.

*Trace analysis.* Once trace links are established, these tools permit the user to create views of a document showing incoming and/or outgoing links (by unique identifier, or by the content of the link itself) in the margins separated from the main document text. Of course views of the trace matrix can be created, but one quickly realizes the limited value of the printed matrix compared to the use of the trace information within the tool itself.

Most tools allow clicking on a link in one document to launch the document it links to, and positions the screen at the linked object. This kind of on-screen capability greatly facilitates the analysis of traces for completeness, accuracy, redundancy, consistency, and ambiguity. (Remember those attributes from review requirements?)

*Change impact.* When a link has been established between a parent and child object, presumably the creator and reviewer(s) of the trace are satisfied with the completeness, accuracy, and so forth. What happens when either the parent or child objects (or both) get changed after the link is established? The validity of the link is now in question, and should be reexamined. To protect reviewers from the need review, rereview, and re-rereview all of the traces each time a document on either end of the trace changes, requirements management tools automatically highlight questionable links; that is, links for which an object on either end of the link has changed. This obviously increases the productivity of the trace reviews.

The tools also allow metrics of such changes. For example, as the software requirements move from version 1 to version 2, a requirements management tool can provide a count of how many trace links to the system-level software tests have changed. That can provide an input to estimating how much work will be required to modify the tests procedures for the changed requirements. These tools are so effective for these kinds of metrics that they have fallen victim to the "shoot the messenger" response of some users. They drop the use of the tool because they can not deal with the bad news the tool delivers (even though it is accurate)!

## Trace Mapping

We have discussed trace resolution above related to the resolution at which trace labels are applied to requirements. Let us spend just a little time discussing trace mapping. Trace mapping refers to the shape or structure of the trace schema at a level more detailed than the schemas used as examples above.

The questions to be answered with mapping selection is, “How many design elements should a requirement trace to?” or “How many requirements should a design element trace to?” Naturally, one could substitute any two design outputs that trace to each other in those questions. We are thinking at the individual requirement or object level in this discussion.

There are four basic morphologies to consider, as shown in Figure 9.12. The preferred mapping often depends on the nature of the two design outputs that are linked. For example, consider the trace from system requirements to software requirements. A many-to-one mapping (many system requirements mapping to one software requirement) would not make sense. There would be more detail in the system requirements than the software requirements. However, consider the trace links from software requirements to system-level tests. In that case, sometimes it might be appropriate for two or more software requirements to be tested in the same test procedure.

In general (and there are always exceptions to the rule), a one-to-many mapping is appropriate for design documents as they become more detailed. A one-to-one mapping (or inferred trace) works well from detailed design to source code. In Chapter 13 it will become apparent that mapping trace links from requirements to test procedures has occasion for one-to-one, one-to-many, and many-to-one mappings. In cases where there is a “many” involved in the mapping, the “many” should not be *too* many. As too many requirements are tested in a single procedure (many to one), the completeness and accuracy of the test is difficult to ascertain during review. A single requirement that requires very many procedures to test it (one to many) is also difficult to review and understand. This is likely the result of a bad requirement that should be broken down into simpler requirements that could be tested satisfactorily with simpler trace mappings.

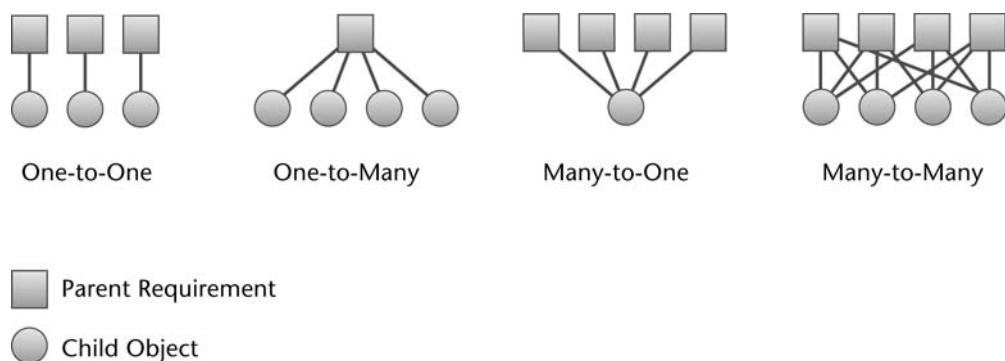


Figure 9.12 Four basic trace morphologies.

Many-to-many mappings have not been mentioned because they are of such limited usefulness. They are so complex it is nearly impossible to use the traces for any type of analysis or metrics. When a many-to-many mapping is necessary, it is usually because of structural organization problems in the documents that are being traced. In these cases, a problematic trace is simply an accurate portrayal of problematic design outputs.

### Can Traceability Be Overdone?

The short answer is yes it can. Traceability seems to be one of those things that people resist until they finally see the benefits. Then, they seem to go overboard and want to trace everything to everything. Often, this happens right after the company adopts a new requirements management tool. I will admit it: our company fell victim to this temptation. I have also observed this behavior on client projects. One client even complains that maintaining the trace is the bottleneck in their product development (and maintenance) processes that limits their productivity.

This is, in my experience, easily solved. For each class of trace between documents to be created, one should ask, “What information do we hope to get from this trace?” or, “How do we plan to use the information we get from this trace?” If there is not clearly definable information or a clearly identifiable use for the information, there may be limited value in creating and maintaining the trace. Especially with a new tool in hand, there is a temptation to trace information simply because you can, or it seems right. No matter how right or easy it seems, there’s no sense investing the time if you cannot describe why you need the information!

That is the last of the activities that span the software development/valid life cycle (often called supporting processes or activities) that will be discussed in this text. These activities are in no way second-class activities or subservient to those mainstream activities in the life cycle. In fact, one could make the argument that because these activities are so prevalent, crossing phase boundaries and disciplines, that they almost dominate the mainstream activities.

We now will transition into describing the mainstream activities. Although the supporting activities may not be mentioned much in the context of the mainstream, the importance of the role of these supporting activities cannot be emphasized enough.

## References

- [1] *Design Control Guidance for Medical Device Manufacturers*, FDA, March 11, 1997.
- [2] IEEE Std 1028-1997, *IEEE Standard for Software Reviews*, Institute of Electrical and Electronics Engineers, Inc.
- [3] AAMI ID26:2004, *Medical Electrical Equipment—Part 2: Particular Requirements for the Safety of Infusion Pumps and Controllers, Section 51.109*, 2004.



**PART II**

## Validation of Medical Device Software



# The Concept Phase Activities

For now, put aside the discussion about the shortcomings of the traditional waterfall life cycle model, and recall that one of the advantages of the waterfall model was the simplistic intuitiveness of the model. We will exploit the intuitive appeal of the traditional waterfall life cycle model to explain the software validation activities that would be an appropriate in each phase of that life cycle. The phases that we will discuss are the concept, requirements, design, implementation, test, and maintenance phases. If a life cycle is chosen other than the traditional waterfall, it will be left to the reader to distribute the activities organized here by waterfall life cycle phase into the appropriate phase of the life cycle that is chosen.

## The Concept Phase

If the device being developed is a totally new device, or a new generation of a legacy device, the concept phase will be more crowded with activities and likely take longer than for a project that simply adds new features to an existing product, or fixes problems that somehow made it to the field. We will not attempt to separate the concept phase activities into groups that are appropriate for various types of projects here. That will be left to the intelligent reader. We will discuss the activities that would be appropriate for developing a new product from the ground up.

So what is the concept phase? Unfortunately, development teams often skip over this phase (and several others) and jump right into implementation. Validation teams often are not called upon until well after the concept phase is history. It is unfortunate because not taking this phase seriously could have serious consequences later. Even more unfortunate is that it (at least to me) is really the most exciting and interesting phase of the device life cycle.

This phase is so interesting because it is the “honeymoon” phase of the life cycle. The whole future of the device lies before the team in this phase. The device can be whatever the team makes it. It is all optimism, nothing has gone wrong yet, and the team is working as one.

That's enough metaphors. Seriously, the objective of the concept phase is to define the product or device that meets the users' and other stakeholders' needs. As the software team moves into the next phase of software requirements, constraints are already in place as the software requirements are designed to meet the requirements set forth in the system requirements (which are the design output of the concept phase activities). There is still room for some creativity later on, but never again in the life cycle is there as much opportunity as during the concept phase. There are a number of planning, and other supporting activities that take place during the concept phase. However, the dominant activities in this phase are to discover the

problem(s) and needs to be addressed by the device, and craft system requirements that lock in a solution based on factual stakeholder needs.

## Regulatory Background

This will be uncharacteristically short because there is little in the form of regulatory requirements for the concept phase. The design controls do refer to design inputs in 820.30(c):

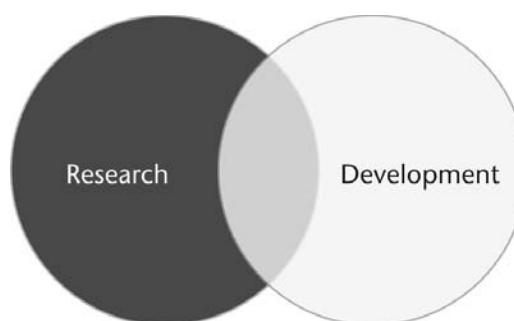
- (c) Design Input. Each manufacturer shall establish and maintain procedures to ensure that the design requirements relating to a device are appropriate and address the intended use of the device, including the needs of the user and patient.

How design inputs (product requirements, design requirements, device requirements, or system requirements are all substituted synonymously here) are determined is not addressed in the regulation. Other than requirements for planning, there are no regulations that would apply to any activities before the above requirement (which actually is a requirement for procedures for the creation of the design requirements).

No regulation? Is that an oversight? It probably is intentional, for good reason. The reason has to do with the difference between research and development. Figure 10.1 summarizes the relationship between research and development. The activities that begin with parsing system requirements into electrical, software, mechanical, and other requirements and end with testing are clearly development activities. The system requirements are being *developed* into the product.

The *research* activities suggested by the figure are the “magic” that happens in the discovery of the system-level requirements for the device. There are many organized methods for eliciting needs from various stakeholders, but much of research is pure creativity, innovation, experimentation, and testing of concepts. To attempt to regulate that would be to stifle the creativity that is so important to research. Furthermore, it would result in mounds of documentation for concepts that ultimately are rejected.

At the intersection (the gray zone of Figure 10.1) is where research and development intersect. Creation of the system requirements document itself is a good example. The system requirements specification (SyRS) is controlled, traceable, and



**Figure 10.1** The overlap of research and development.

should be reviewed. The activities around the SyRS look for all the world like development activities, yet the inputs used for this design output come from largely unregulated and less controlled activities.

Research in the context of needs identification deserves a few more words of explanation. Research in this context does not necessarily mean the white lab coats and test tubes we often associate with our image of research, although it could. In addition to researching new sensors, treatments, diagnostic methods (i.e., pure scientific or engineering research), research in the concept phase of a device might include user research into needs, opinions, or capabilities that are accomplished through interviews, surveys, or observations. Early prototype devices might be tested with intended users to perfect the user interface or other attributes related to human factors engineering. Prototype research system components are not part of the development process, and can be produced without prescribed requirements and designs because, after all, their purpose is to help discover the requirements. Should the results generated by the research prototypes and their testing be documented at all if the regulations do not require it? Of course. It is simply good engineering practice to do so, and it leaves a record later in the life cycle of just where those system requirements came from and how they were produced.

Some readers are now thinking that this is just the loophole they were looking for to justify old habits of trial-and-error engineering with no design controls. Well, you aren't the first to think of this. I probably meet with half a dozen companies a year that think they have discovered a way around all this pesky design control stuff. They will prototype and test until the device is implemented, then go back to create documents to make it look like it was a controlled design process that got them there. I'm not fooled. Investigators often are not fooled. These clients didn't save time because they developed their product this way. They certainly did not come up with a better quality product, and they have exposed themselves to a regulatory risk if their falsification of documentation is discovered.

## Why a System Requirements Specification Is Needed

Think about why formal system requirements may be beneficial to the design and validation efforts. There are three major purposes for a system requirement specification (SyRS):

1. The SyRS (and all design documents) serves as a means of communication to the entire product team. It is the one central location to which anyone on the team could refer back to understand the stakeholder need behind any other requirement, design element, software, or test that follows. It is the one place that changes in stakeholder needs must be documented, and from which the trace links will lead the way throughout the system design to make those changes.
2. The SyRS is the basis from which the next level of requirements will be allocated and developed. A single system-level requirement may easily link to requirements documents for a number of disciplines that will contribute to that single requirement. A single SyRS requirement, acts to coordinate the efforts among those disciplines working in concert to fulfill that need.

As an example, consider this system-level requirement, “The user interface (UI) shall be viewable by users with normal (20/20) vision from a distance of 10 feet at an angle of +/- 45 degrees horizontally and +/- 20 degrees vertically, in ambient light conditions from total darkness to bright sunlit rooms.” Granted, this requirement is compound and is not as quantitative as it could be, but to make a point, the SyRS requirement will impact software (assuming the UI is controlled by software) by affecting font sizes and types, color/contrast selection, and control over the background illumination levels. The system electronics will be affected by the choice of display size, and choice of background intensity capability, the viewable angles allowed by the display technology, and even the power available to light the display background. The mechanical design will be affected by the visibility angles allowed by the enclosure, and the placement of knobs, switches, and cables that could obscure the display. Without the SyRs requirement, the three discipline teams could be working totally independently to create a perfectly implemented engineering design that would not work in the intended use environment.

It should go without saying, but let’s be explicit. The need for an SyRS as a communication tool, or a basis for requirements development means that the SyRS is developed before any of the specific requirements specifications, and is maintained throughout the life cycle. The value of the document is greatly diminished if it is an end of project afterthought. Treating the SyRS this way is little more than retro-documenting the device the way it ended up, and is not really indicative of the real needs of the user.

3. The SyRS is the basis for validation testing. Verification activities like reviews and verification testing are good for monitoring engineering accuracy; that is to say that verification activities assure that the design outputs for each phase accurately move the design forward as prescribed by the design inputs of that phase or step in the design process. However, there is a need to validate that the final product meets the needs of the design inputs of the system (i.e., product).

Why? Because the design can drift over the course of months or years that it takes to develop a product. Verification can be short-sighted, leading the team to believe they are meeting intent from the inputs with which they are provided. Yet it is quite possible to perfect the engineering design of a device without meeting the needs of the intended users. The system requirements are the record of requirements needed to fulfill the needs of all the stakeholders interested in the device. Validating the fulfillment of those system requirements is final confirmation that the device is, indeed, the device that was defined by the design inputs.

## Validation Activities During the Concept Phase

As mentioned earlier, the main activity during this phase, or at least the main design output that results from the majority of effort in this phase is the system requirements specification. Is the SyRS the responsibility of the validation team? In my

opinion, it is probably beyond the scope of responsibilities of the software validation team. However, more than one company has placed responsibility for the SyRS (and most other documentation) on the validation team. This sometimes seems to be almost a punishment for pointing out the need for system requirements. Certainly the validation team should have a big role in the creation and review of the SyRS to be sure it is traceable, testable, and accurate. That role may be to carry the responsibility for its creation and maintenance. However, for a project and device to be successful, the input and assistance of many stakeholders including designers and developers is required.

What the validation team must concern itself with is verifying that safety, efficacy, reliability, usability, manufacturability, serviceability, salability, security, and privacy are adequately described in a robust set of system-level requirements for the device whether or not they are responsible for their creation. Additionally, and just as importantly, the validation team must verify that those requirements are specified in a way that can be verified and/or validated at appropriate phases of the development life cycle. Since many system requirements are not determined to be software requirements in this phase, the validation team needs to think outside the software sphere and work to make all requirements equally useful, not just those that are anticipated to result in software requirements.

In addition to the SyRS, other validation related activities include:

- Preliminary risk analysis and risk management activities. It is amazing how much information can be gleaned from preliminary risk analysis activities. Clinical specialists have great insight into the potential risks. Corporate experience with similar products can yield a large number of risks in the very first risk analysis meeting. Research into problems that competitive devices have had in the market is also a good source of identifiable risks. Why start in the concept phase? Simply because it will never be easier or less costly to identify risk control measures that become design requirements.
- Planning for verification and validation, risk management, defect and issue management, and configuration management are all possible, at least in an early form, in this phase.
- Reviews of any plans created in this phase;
- Review of the system requirements specification itself;
- Issue/defect management should be initiated for issues raised and problems found in reviews. We have found that starting this activity early is the best way to keep track of issues for the life of the project. Yes, there is some overhead in tracking issues. However, it would be difficult to argue that it would be more efficient not to track issues, raising them repeatedly throughout the development life cycle, and risk forgetting important issues until late in the life cycle when they are most expensive to address. The level of formality in early phases like the concept phase may somewhat reduced from formal defect management in final test.
- Preliminary configuration management should be started even in the concept phase. A number of stakeholder needs documents could result from research done in this phase. Since they are inputs to the SyRS, keeping the evolving

needs documents in synchrony with the SyRS is no less a task in this phase than during the later purely development phases of the project.

Another set of tasks related to configuration management relate to understanding and planning for code ownership and control at the end of the project. This is reasonably straightforward when all of the software (i.e., code) used in the device is custom-developed by the device manufacturer. It is less clear when software is purchased from commercial sources, downloaded from free sources on the Internet (a.k.a. software of unknown provenance or SOUP), custom-developed by contract development companies or individual contractors, or is open source software. In each of these scenarios, the device manufacturer has given up some control of the software, but retains all the regulatory responsibility for the software as used in the device.

## Make or Buy? Should Off-the-Shelf (OTS) Software Be Part of the Device?

In addition to the design history file and intellectual property ownership considerations hinted at above, one should consider the level of confidence one has in any OTS software that may become an integral part of the device. If confidence is on the low side, one should determine as early as possible whether it will ever be possible to establish confidence in the software ... and if so what level of effort will be required to do so. Sometimes OTS is just a bad option and the software should be created and controlled by the medical device manufacturer. It is better to know that early than to discover it when problems surface after release of the software.

Why should the origin of the software be addressed in the concept phase? The origin and “pedigree” of the software should be researched as early as possible because bad choices are easier to correct early in the life cycle. Imagine finding a defect in a communication protocol during final testing. If that protocol was freeware SOUP and no source code is available, there are no good alternatives to a costly redesign late in the game.

Why should the validation team be concerned about where the software comes from, or how much control the device manufacturer has over that software? First, and most obvious is that the validation team will be responsible for “validating” that software. Some confidence that the software is developed by reasonably skilled and experienced engineers under some modicum of control would be wise, and requires some preplanning, investigation, and decision-making on the selection of sourcing externally sourced software. Once the decision is made to use software from external sources, those responsible for validation must consider how the software will be validated for its intended use in the device.

Testing of externally sourced software (i.e. software acquired for use) is possible to some extent and is covered later in Chapters 13 and 19. Other considerations related to validation of this software *other than testing* include:

- *Defects:* Does the software supplier provide information about defects in the software? How do you get that information; from a Web site, or do they send notification? Who pays for the repair of any defects? How often do repair

patches or updates get issued? Infrequent updates may be a sign of unresponsiveness. Frequent updates could be a sign of immature software that will be just as problematic.

- *Documentation:* Are requirements or specifications available at a level of detail that is testable? Are instructions for use, limitations, assumptions, and performance specifications available, or will this be a discovery process?
- *Test results:* Has the software been formally tested by the supplier? Are the results and test procedures available, at least for review to determine the depth of testing? Are the procedures available in case you need to further test future versions?
- *Malware:* Are the software and the distribution media always checked for malware before distributing software, patches, or updates?
- *Availability of source code:* How can you track down potential problems if the supplier is unavailable or unwilling? Remember that the medical device manufacturer is legally responsible for the software in the device. You will need assurance that the source code is in your possession, in escrow, or that you have contracted assurances that the supplier will be responsive to future problems.
- *Availability of supplier:* Availability of the source code is of prime importance, but a close second is the availability of the supplier, and specifically the developer. Your chances are good with commercial OTS providers with recent software, but they lose interest when the software is several versions old. Your chances are poor with independent contractors who may be writing your software until they find a full-time job, or between extended vacations to the Caribbean. With SOUP, since the provenance (i.e., pedigree or source) is by definition unknown, it is very unlikely you will find the developer to fix problems.
- *Ownership:* Who owns the source code? Ownership of the software is not always clear-cut, especially for contract developers who make development contracts attractive because they “already have a lot of that code that was developed for other clients.” You could end up not owning the software embedded in your device because it may have originally been written for someone else—even a competitor. Equally bad, the software you paid to have developed could end up in a competitor’s device.

Chapter 19 covers alternative methods of validation for software that is acquired for use (i.e. OTS). Many of those methods are covered in the preceding bullets. One might ask why so many validation activities are suggested for so early in the lifecycle. There are two good reasons:

1. Because it is possible. The OTS software is by definition completed and validation can be considered.
2. There is a good possibility that considering the above concerns could lead to a conclusion that OTS is not a good option, or that a particular OTS choice is not a good option. Common sense would indicate that it is better to know that early in the life cycle rather than later when many assumptions and designs already depend on the software.

## The System Requirements Specification

The development of the SyRS is the primary design output of the concept phase. There is generally a desire to keep the SyRS as short and simple as possible. This increases the likelihood that the large group of stakeholders who have an interest in the specification will actually read it and understand it. However, technical usefulness should not be sacrificed in the interest of brevity. It is not uncommon for a development team to underestimate the importance of this document, and dash off a two-page SyRS just to check it off the list of deliverables for the design history file.

In this section we will focus on some of the factors that contribute to the success of a system requirement specification (SyRS). In particular, we look at the who, the what, and the how: This is a good approach for all of the design and validation:

- Who the intended audience(s) is/are for the document and what they need from the document;
- What needs to be in the document that satisfies those needs;
- How that information can be gathered, processed, and communicated to the intended audience to satisfy their needs.

### Who Is the Intended Audience?

First, what do we mean by an “audience”? For our purposes, the audience simply will imply a group of people who will have reason to read the SyRS, although their motives for reading the document may be very different. This provides a good segue into thinking about the classes of readers of the SyRS and why they might be interested in its successful implementation. The audience is comprised of two main classes that we will call the “customers” and the “solution providers.” (Understand that “customers” in this context include the actual commercial customers for the device, as well as other interested parties. It is an ambiguous use of the term, but unfortunately has become common usage in this context.)

The customers are those who have expressed a need for the device or have a problem that will be solved by the device or some specific element of the design of the device. For example, intended users, patients, clinicians, marketers of the device all have specific needs that need to be fulfilled by the design of the device. Other groups are less often overlooked as customers of the SyRS such as the legal team (opinions about patent strategy to be observed in the design), quality team, and the regulatory team (opinions company liability and about design elements that cannot change without violating the regulatory submission strategy). The customers have an interest in being part of the audience for the SyRS to assure that their needs are accurately represented in the document that defines the device.

The solution providers include the developers for the device such as the electrical, mechanical, and software engineers as well as scientists, testers, and human factors engineers. Their interest in being part of the audience for the SyRS is more technical in nature. They want to assure themselves that an adequate level of detail, accuracy, completeness, and testability are present for them to understand the specific requirements of their respective specialties. The solution providers are

“consumers” of the information in the SyRS. The customers of the SyRS are the source or providers of the information.

Some specialties have an interest in the SyRS as both customers and as solution providers. Examples of this might be the manufacturing and service groups. Both of these groups may act as providers of information; that is, information relevant to the design of the device that will make the manufacture and service of the device easier, less expensive, or of higher quality. On the other hand, the SyRS may well contain requirements that will impact the manufacturing and service groups making them solution providers. For example, requirements related to sterility, shelf life, or cost are likely to impact manufacturing group’s plans as the device design is transferred to manufacturing.

## What Information Belongs in an SyRS?

The contents of any type of requirements document and the level of detail of those contents are often the subject of debate among the development and validation teams. That topic will be given more treatment in the next chapter, but for the SyRS, the amount of detail depends on the situation. For example, the need for and choice of operating system for a hand-held device with embedded software definitely leans toward being a design decision, not a software requirement or a system requirement. On the other hand, a device that is a software product for home use that runs on the user’s home computer would definitely have a requirement for what operating system(s) it would run on (i.e., PC or Mac). That requirement would definitely be a system-level requirement. Where a requirement goes in the requirements documentation hierarchy depends more on the flexibility of the design around that requirement or when the information is available to specify the requirement, than it does on the type of information. That is partly why the “Is this really a requirement?” argument often takes place in requirements reviews.

For now, we will focus on what types of information should be found in the SyRS, and leave the discussion for how much for a later time. As has been already mentioned, the SyRS is the definition of the system (i.e., the device). It will become the key document used for validation testing to confirm that we have designed the right product. The FDA’s GPSV (Section 3.1.2) partially defines validation as, “Confirmation by examination and provision of objective evidence that software specifications conform to user needs and intended uses....” For that confirmation to take place, we need well-established user needs and intended uses. That is the role of the SyRS.

### Intended Use

Many of the activities that lead up to the actual authoring of the SyRS are related to collecting user needs and distilling them down into intended uses. So what is intended use? Let’s consider a syringe pump. Initially the intended use might be penned as “...to deliver medication from a syringe to a patient.” Is that adequate? Does that tell us anything that would be useful in validating for conformance to

intended uses? Probably not, since almost anything would pass a test of that intended use.

What else might be useful in providing enough information for us to devise a test to determine whether the resulting device is adequate for its intended use? That would be determined by the user needs, or more generally the stakeholder needs. Perhaps marketing had a need for a syringe pump to be used for the precise administration of anesthetics from a syringe. That narrows the field a bit, but raises new questions. For instance, is there anything special about pumping anesthetics, or about the environment in which such a pump would be used, or about those users who might be charged with the responsibility of use of such a pump? As one starts to explore such questions, one is often lead to more and more detailed system level requirements that are needed to meet a generalized need.

### Intended Users

In exploring who the intended users of a device will be, one is often lead to the realization of more and more detailed requirements for the device. For example, language, experience, education, culture, age, and disease state are all examples of attributes of users that would impact the design of the device and the validation of conformity to user needs. For example, consider a glucose meter intended for home use. The intended users certainly will include diabetics, and among those users will be some who suffer from diabetic retinopathy and neuropathy. How will that impact device design? Clearly, small buttons and small displays are going to be a problem for the intended users of this device. The consideration of the intended users would lead to a series of system level requirements that would be needed to fulfill the needs of the intended users.

### Intended Use Environment

The environment in which a device is intended to be used is almost always different from the environment in which it is developed, and in which most of the verification testing is conducted. An example list of things to be considered for intended use environment is given here. It is not intended to be an exhaustive list. It is a list that is provided to get the reader to think critically and creatively and to create similar lists that are relevant to his/her own specific device.

#### *Physical*

Consider the physical environment in which the device is intended to be used.

Will users be gloved?

Will the user interface need to be sanitized or sterilized?

What are the ambient light levels, how far will the device be placed from the user?

Are there specific viewing angle requirements?

Will it be wall- or pole-mounted, placed on the bed or tabletop, hand-held, or a combination of the above?

What are the power requirements?

Are there international power requirements?

Are there battery power needs, battery replacement or recharge requirements?

What kinds of electromagnetic radiation, moisture, temperatures, atmospheric pressures, chemicals, and dirt will it be exposed to? Is it an explosive or flammable environment? (e.g. some anesthetics, oxygen)

As an example, one of our clients manufactured a heat ablation device that seemed to have a good field experience in all of their markets but China. China's equivalent of the outpatient clinic in which the procedure was performed was often not heated, or not heated above 50 degrees in those days. The device could not produce enough energy to heat the probe effectively in that ambient temperature. The device was designed and tested in a near 72-degree or warmer environment. The intended use environment need was not discovered until the device was complete and failing in the field.

### Standards Environment

Many medical devices are subject to industry standards or regulatory guidance documents that are specific to the device type. This was discussed under the topic of traceability in Chapter 9. If it is necessary for a device to conform to standards or guidance, then the requirement for that conformance should surface early in the SyRS. As mentioned in the traceability discussion, individual standards requirements might be embedded in the SyRS (if they are few), or it might simply require conformance to the relevant standards or relevant sections of standards. Do not underestimate the magnitude of this task! Some devices have many applicable standards and the relevancy is not a trivial issue. It is a fool's errand to embark upon a design and development project without a full understanding of what the standards and regulatory guidance will require of the end result. Many a device has been redesigned at great cost, delay, and heartache because standards were not mentioned or studied until the design was nearly complete.

### Legacy Environment

If the manufacturer is developing a new version of a device already on the market, there is the potential that a user may have used the predecessor device or may still be using older devices when the new device hits the market. (After all, the most likely new customer is an old customer.) This should be taken into account in the SyRS to avoid any possibility of context confusion for users that may need to switch between the two user interfaces often, and under stressful conditions. Changing terminology between devices, drastically different control placement, reorganized displays, and similar but different disposables can all lead to these kinds of problems. The same kind of context confusion is possible with competitive devices, and should be considered even though it is much different problem. This is not to say a manufacturer should never change a device's look and feel. That would permanently prevent improvement. However, the possibility of confusion with other devices should be considered and measures taken to minimize that possibility.

## Platform Environment

Computer-based devices (i.e. not embedded software), or devices whose user interfaces run on a connected computer, need to consider the platform requirements for the physical computer and operating system. The sourcing of the platform elements (i.e., do the users buy their own PCs or does the device manufacturer provide them), and control of change to those elements are important configuration management considerations. Being specific about the platform environment early in the SyRS allows the device design to adjust to the risks of these system requirements.

## Organization

A SyRS organized around needs, users, or use environments might work, but more often they are organized around the system (i.e., device) features, functions, or operating modes with the requirements that have been distilled from the needs reorganized accordingly.

System requirements should include performance specifications as often as possible. It is not unusual for a requirement from a needs analysis to be rather qualitative such as “user friendly” or “the display must be large enough to see.” These are valid user needs. They deserve further exploration by those creating the SyRS. The users would not have mentioned these as needs if there were not some underlying concern or past problem. The challenge for the SyRS is to refine that user need into a set of requirements for the SyRS that are more descriptive and are ultimately testable. For example, the above “large enough to see” need could be broken down into SyRS requirements for distance from the viewer, viewer’s visual acuity, viewing angle, ambient light conditions, and so forth.

## How Are System Requirements Gathered?

A user need statement might be very different from a system requirement. This is because the user (or other stakeholder) thinks in terms of their problems, not in terms of the specifications for the solution to their problems. The latter is the responsibility of the writer(s) of the SyRS.

Communication between the two audience groups (customers and solution providers) often requires different techniques and communication media. The solution provider group may prefer detailed text-based, traceable requirements as the medium most useful for them. The customer class may prefer verbal communications, pictures, flow charts, and early prototype examples. As a result of this, it may be beneficial to think in terms of needs documents in the language of the customers, and the SyRS in the language and format most useful to the technical solution provider class.

How does one go about determining user needs? There are a variety of ways, and there are experts who spend careers on perfecting techniques for this. There are interview techniques, focus groups, surveys, observational skills, and so forth. Regardless of the format chosen, those stakeholders with needs to be met by the new device probably are not going to be able to articulate those needs without a lot of coaching or prompting.

Asking “What are your needs for this device?” will probably get a few items, but more likely will get a blank stare from the prospective user. Instead, more specific but open-ended questions are likely to lead to much more interesting dialogs from which the interviewer can later isolate individual needs. For example:

Interviewer: Where do you see this device being used?

User: Primarily in the operating room

Interviewer: So there are other places than the operating room?

User: Why yes, we could use the device in transporting critical patients between hospitals also.

Interviewer: Is that by ambulance?

User: Usually, but also by helicopter sometimes.

Interviewer: Is that all?

User: Well we do have that mobile unit that goes out to the remote parts of the county.

This kind of give and take could go on for pages, but look at the number of needs that were identified for location of use. Each of those could be explored for needs specific to the location. Had the interviewer only asked “So what are your needs for this device?”, he would have received an answer along the lines of “It just needs to work reliably.”

What do good system requirements look like? They share many of the attributes of other requirements (like software requirements) of being complete, accurate, unambiguous, traceable, and testable. This has been discussed some on the topic of reviewing requirements in Chapter 9. It will be discussed in more detail in the next chapter on the requirements phase. However, there is one thing that is different about system requirements. They need to be somewhat abstract. They need to be terse descriptions of the requirement as interpreted from the user needs but devoid of any indication of a preferred solution. It is tempting to let design creep into a SyRS, and usually when it does it is for honorable motives. If the SyRS author already knows what he or she wants, why not just say so? The reason is that maybe that idea is not the best idea. Maybe that idea is not achievable given other constraints or needs to be fulfilled. Locking in a design preference as a system requirement can unduly bias a design by limiting alternative solutions.

## Further Reading

Let’s leave the concept phase at that. There are many books and experts on the techniques that were but only briefly mentioned here. There are also many books and publications on writing system requirements. A few of the more useful, or at least more enjoyable are listed in the Select Bibliography.

## Select Bibliography

Norman, D. A., *The Design of Everyday Things*, New York: Basic Books, 2002.

Wiegers, K. E., *More About Software Requirements*, Redmond, WA: Microsoft Press, 2006.

Wiegers, K. E., *Software Requirements*, Redmond, WA: Microsoft Press, 2003.



# The Software Requirements Phase Activities

The following was heard by a project manager on the same day about the same software requirements:

Software Quality Engineer (Tester): “These requirements are worthless. They don’t contain enough detail for me to write detailed tests. I either guess how the software will work, or simply write a test that verifies that it works they way I observe it to work. Both seem like a waste of my time, and I’ll certainly have to change them a lot as I discover how the software was really supposed to work—if I ever do. This is feeling like a big waste of time.”

Software Development Engineer (Programmer): “I don’t really need these software requirements. I already know what to implement. I have it all in my head and in pieces of code I’ve already written. We’re on a deadline. I don’t have time or interest contributing any more to it, or even in reading it. Besides it contains too much design information that is going to limit my ability to make changes and is stifling my creativity.

How can it be that the same software requirements can be lacking in detail yet be perceived as being too detailed and restrictive at the same time? The perspective of the reader, the needs of the reader, and the historical discipline (or lack thereof) within the organization all are likely contributors to this paradox. The tester needs the requirements as inputs to understanding how the software is intended to work. The programmer should also need the requirements as inputs to understanding how he should implement the software. Evidently, the programmer feels that he can go on a creative binge without regard to the requirements. Most likely he feels that way because historically he has been allowed to get away with that lack of engineering discipline.

Does that sound a little harsh to the programmer? Maybe so, but this is a book on validation after all. Imagine if the roles were reversed, and the tester quipped:

“I don’t really need the software requirements. I already know what tests I want to write. I am on a deadline, and do not want to get bogged down with requirements when I should be writing tests. Besides, these requirements are too detailed and are going to restrict my ability to test this product the way I want to. The programmer can just change the software so that it passes my tests later on.”

That sounds a little ridiculous, doesn’t it? However, why should that be any more ridiculous than the statement of the programmer in the opening dialog? It

really is not more ridiculous, but it is likely that the programmers historically have been allowed to get away with this type of “professional” behavior.

## Introduction

It should come as no surprise that the focal point of all activities in our software requirements phase is indeed the production of the software requirements. The verification and validation activities in this phase are primarily responsible for verifying that system functionality assigned to the software for implementation is interpreted correctly in the software requirements. The software requirements specification (SRS) is the design output of this phase activity. The system requirements specification (SyRS) is the primary input to the creation of the SRS in this phase.

This of course assumes that all the desires and needs of the prospective users and other stakeholders have been accurately represented in the SyRS to start with. Traceability from the SyRS back to the needs statements provides a valuable tool for interpretation of individual requirements in the SyRS that inevitably will be needed in later phases of the life cycle. Risk control measures that are identified during the preliminary risk management activities of the concept phase may be traceable to the SyRS, or may trace directly to requirement specifications such as the SRS, or the equivalent of the SRS for electrical, electronics, mechanical, and so forth.

If verification of the SRS (assuring that SyRS requirements are represented accurately) and validation of the SRS (assuming that these software requirements do in fact represent adequate fulfillment of the user needs) were all that were needed in this phase, this would be a short chapter. However, since the software requirements are of such key importance for all of the activities that follow in the software development and validation life cycles, it is important to be able to distinguish good requirements from bad. Those responsible for verification and validation of requirements must be able to review those requirements to be sure they are accurate, complete, unambiguous, consistent—and testable. Also, for unfortunate reasons seen in the dialog at the beginning of the chapter, the authorship of the software requirements often is made the responsibility of the validation team simply because the development team does not want to take part in designing the requirements for the software they will be charged with creating. Therefore it is important for those charged with verification and validation to also be well-trained in the skills needed to create, or at least recognize, good software requirements.

## Regulatory Background

We will start this chapter by taking a brief look at what the FDA regulations and guidance documents have to say about requirements. From the FDA’s GPSV:

### 3.1.1 Requirements and Specifications

While the Quality System regulation states that design input requirements must be documented, and that specified requirements must be verified, the regulation does not further clarify the distinction between the terms “requirement” and “specifica-

tion.” A requirement can be any need or expectation for a system or for its software. Requirements reflect the stated or implied needs of the customer, and may be market-based, contractual, or statutory, as well as an organization’s internal requirements. There can be many different kinds of requirements (e.g., design, functional, implementation, interface, performance, or physical requirements). Software requirements are typically derived from the system requirements for those aspects of system functionality that have been allocated to software. Software requirements are typically stated in functional terms and are defined, refined, and updated as a development project progresses. Success in accurately and completely documenting software requirements is a crucial factor in successful validation of the resulting software.

A specification is defined as “a document that states requirements.” (See 21 CFR §820.3(y).) It may refer to or include drawings, patterns, or other relevant documents and usually indicates the means and the criteria whereby conformity with the requirement can be checked. There are many different kinds of written specifications, e.g., system requirements specification, software requirements specification, software design specification, software test specification, software integration specification, etc. All of these documents establish “specified requirements” and are design outputs for which various forms of verification are necessary.

The first of the quoted paragraphs, in addition to emphasizing how crucial requirements are to successful validation, also recognizes the requirements are difficult, if not impossible, to create accurately and to an adequate level of detail in one pass. The recognition that requirements are “defined, refined, and updated as a development project progresses” indicates that the FDA clearly understands that it is unlikely that a project team will be able to adhere strictly to a traditional waterfall life cycle model.

The second paragraph clarifies the relationship between requirements and specifications, but the second paragraph, which itemizes some of the “many different kinds of written specifications,” goes on to say that each of these specifications establishes “specified requirements.” Later in this chapter we will touch on a common controversy over what constitutes requirements and constitutes design. At least from the regulatory perspective, it would appear that system, software, design, and test specifications could all be made up of “requirements.” Consequently, although the philosophical debate over requirements versus designs is interesting to developers and testers, the FDA treats them all the same. The concept of a requirement is a relative term not an absolute term. A requirement here is viewed as simply an input to an activity that results in some level of design output.

The GPSV goes on to mention requirements in the context of software validation:

#### Software Validation

... Since software is usually part of a larger hardware system, the validation of software typically includes evidence that all software requirements have been implemented correctly and completely and are traceable to system requirements.

—*General Principles of Software Validation, Section 3.1.2*

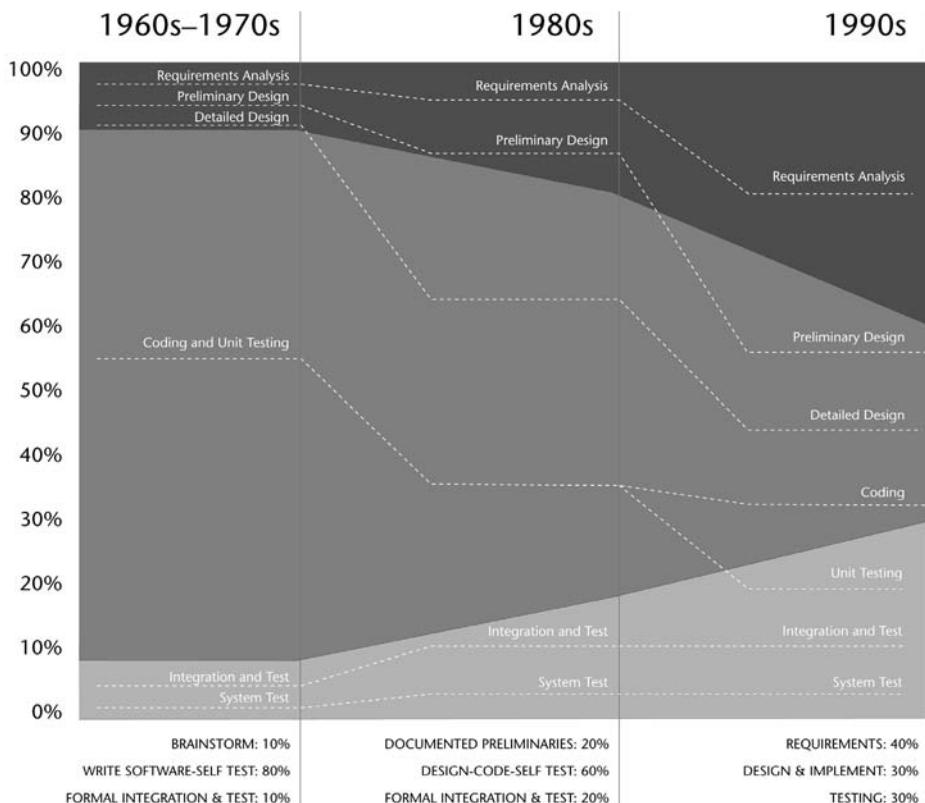
The evidence that all software requirements have been implemented properly, coupled with the understanding of the FDA's relative requirement concept, gives this paragraph a meaning beyond what many in the industry have traditionally interpreted it to mean. The traditional interpretation has been that traceable evidence of verification to a specific document full of software requirements is required. However, it is not uncommon for project teams to manage several documents at a predesign level, only labeling a subset of them as "requirements" to avoid having to trace to the nonrequirement documents and/or verify their correct implementation. Since the GPSV in Section 3.1.1 considers any description of the software as a requirement, it appears that they all should be verified. With that understanding, there is little reason to parse the language to hide requirements from validation. It would seem to make this requirements shell game contrary to the regulatory intent.

Does that mean that everything in a specification is a requirement? The answer is no. It is common for requirements specifications to contain narrative; that is, text which improves readability and understandability, but is only setting up for the requirements that follow. However, anything that describes the function, behavior, or design of the device is expected to be traceable back to system requirements, and should be validated—regardless of whether they are called requirements or not.

## Why Requirements Are So Important

The National Institute for Science and Technology (NIST) issued a report in 2002 called "The Economic Impacts of Inadequate Infrastructure for Software Testing" [1]. It is fascinating reading, but mostly out of scope for this text. The report covers the topic for software in general, not specifically medical device software. It includes a reference to a study in 1995 by Andersson and Bergstrand [2]. One interesting set of facts from that study is further replicated here in Figure 11.1 showing the allocation of effort to various activities related to software development and validation. The study revealed that software development projects in the 1960s and 1970s spent 80% of the time and effort on writing the code and unit level testing it. (Although from personal experience, I doubt that much of that 80% was spent unit level testing!). Only 10% of the effort was spent in requirements and design activities. By the 1990s, a full 40% of the effort was being spent on requirements, 30% on design, and the remaining 30% shared among coding and integration and system-level testing. These results clearly show that software developers and validation professionals across industries have, on a relative scale, put more effort (by more than a factor of 4) into developing requirements since the 1960s and 1970s. It is safe to assume that development teams are not spending more time on requirements because it slows them down, or because it's a waste of time. In fact, since this study is not specific to the regulated medical device industry, one cannot even make the argument that they spend time on requirements because it is a regulatory requirement. What value do they see in requirements?

In Chapter 1, the evolution of software development was briefly discussed. The software that is embedded in medical devices has grown from hundreds or thou-



**Figure 11.1** Evidence of growing reliance on requirements.

sands of lines of code to, in some cases, millions of lines in today's devices. The development and validation teams have grown from one or two engineers sharing a cubicle to dozens of engineers scattered across different continents. Perhaps it is simply the need for requirements to coordinate these large-scale activities that has driven the increased attention that requirements get in the life cycle. Regardless, let us take a brief look at what requirements can do for software development and validation. That should provide some understanding of why requirements are so important, and further will help demonstrate whether requirements as written are fulfilling their intended purpose.

Remember from the discussion about the regulatory background above, that “requirements” can refer to system-level functional (or behavioral) requirements, software functional requirements, design requirements, or even test requirements. However, for the rest of this chapter, when requirements are referred to, it will be implied that they are functional or behavioral requirements of device software.

So, why should anyone want software requirements and what value do they bring?

1. Software requirements document what the device software should do, or how it should behave. Software requirements are a design input for software design and for system-level software test procedure development.

The benefit: One should be able to create the high-level and detailed-level design of the software using the software requirements as an input. One should also be able to create system-level test procedures based only on the information available in the software requirements. This results in more predictability in the output of the design and implementation activities, relieves programmers of the need to invent-on-the-fly, and gives testers information for developing procedures in parallel with the software design and implementation.

2. The software requirements are a means of communication to those responsible for designing, implementing, and testing the software. They are also a means of communicating to the project stakeholders in plain English how the software will operate once designed and implemented.

The benefit: Technical developers have one place to look for information on what they should be designing. Further, the centralized requirements have been vetted by all stakeholders, so they provide an organized means for communicating with all stakeholders. It should be remembered that an important audience for the software requirements are nontechnical readers. Further, the software requirement document(s) may be the last documents in the design history file that the nontechnical audience will review thoroughly.

3. The software requirements provide traceability back to the source of the system-level requirements and/or user needs.

The benefit: This is important for several reasons. First, this kind of traceability provides a completion metric for the software requirements. The software requirements are complete when each of the system-level requirements allocated to implementation in software is traceable to one or several software requirements. Second, software requirements that do not trace back to system-level requirements are indicative of gaps in the system requirements, or of additional features that crept into the software requirements that were not intended system-level requirements. An additional benefit of having such a metric is that it improves predictability in managing the development and validation project.

4. Formal documentation, version control, and change management of software requirements provides a means for controlling the feature creep to which many projects fall prey.

The benefit: Controlled requirements subject changes to analysis that weighs the benefit of the change with the cost in time and money. Further, the benefits of the changes are also weighed against the incremental risk that new problems will be introduced. Version control tools also provide an excellent means of tracking the history of how the product concept evolved during the course of the development life cycle.

5. Software requirements should be used as a working document for assignment of design responsibility to the development team members. Design activities should be restricted to that functionality documented in the requirements. When the designers feel the need to design additional software functionality not described in the requirements, it represents an opportunity to capture the newly discovered functionality in the requirements document.

The benefit: This is valuable not only to document the actual requirements as implemented, but as additional input to the test team so that they can create test protocols for the additional functionality. If developers simply implement the newly discovered needs without communicating to the remainder of the team, test coverage will be limited or totally missing. Furthermore, the requirements document and linked design document will have nothing to offer future generations tasked with maintaining the software.

6. The software requirements are the basis for system-level testing of the software implemented functionality of the device.

The benefit: This is been touched on in several different ways, but let us be explicit here. If functionality is not documented in a requirements document it is unlikely to be tested by the test team. Untested functionality will have a higher likelihood of being defective than functionality that has been reviewed, critiqued, and tested.

It is not unheard of for software test developers to create test protocols for functionality that is not described in a requirements document, whether the requirements don't exist, or are just sorely lacking in detail. In fact, on occasion, we have seen clients who refer to system-level test procedures as the best place to look for where the intended functionality is described. Several things are wrong with this situation. It is clearly indicative of inadequate software requirements if the test team is forced to create their own requirements within the context of the test procedures. The test procedures are not as effective as they might be if generated from a requirements document because, to some extent, they simply test the *observed* behavior rather than the *intended* behavior. In other words they test that the software is doing what they observed it to be doing. It is little wonder that test procedures based on observed behavior are much less effective in finding defects. Finally, forcing future maintainers of the software to discover the intended functionality (if indeed it is intended functionality) of the software from scattered test procedures is not efficient, effective, or likely to be done.

7. Comprehensive, controlled software requirements provide a means of controlling who is making product design decisions, and by a transparent process that allows for adequate review by all the project stakeholders.

The benefit: Controlled review, revision, and acceptance of all the project stakeholders is the best way to make sure everyone is informed and approves of the decisions being made for the device being developed. Imagine a thin set of software requirements that is lacking in detail and is not complete. That is, it lacks in both breadth and depth. At some point in the life cycle, those vague, incomplete requirements will be translated into software design and/or into code itself. It is at that point in time that the decisions will be made for *exactly* how the software will function. The decision maker will be a software engineer, maybe even a junior-level engineer or intern. The decision probably won't be reviewed by anyone else, if anyone else is even aware of it. That is not how one wants your device behavior decisions to be made.

8. Controlled, detailed requirements provide a mechanism for high-level debugging of the virtual device as described in requirements before costs for design and implementation begin to accrue.

The benefit: Correcting erroneous, incomplete, or misunderstood requirements for software behavior is never less expensive than when the requirements are only embedded in requirements documents. Many development and validation activities depend on the documented requirements in later phases of the life cycle. Finding problems related to requirements late in the life cycle will be much more expensive to repair than at the requirements review stage of the life cycle. Of course, one cannot do a good job of debugging from requirements unless they are sufficiently thorough and adequately documented.

The activities related to the review, revision, and acceptance of requirements are perhaps the most valuable (and efficient) of the verification activities. Using the techniques already described under the topic of reviews, it is typical for review teams to be successful in finding hundreds of problems in an SRS. Sometimes the authors of the specification do not feel that it is much of a success. Of course, managing that many defects becomes a problem of its own, but consider the alternative. The alternative offered by not reviewing or not tracking findings is confusion, rework, and costly modifications as the requirements defects are discovered and rediscovered in later phases of the life cycle.

## **The Role of Risk Management During Requirements Development**

Inasmuch as risk management is a validation activity, there are two specific risk management tasks that are appropriate as software requirements are being developed.

1. The validation team should verify that any risk control measures identified during the concept phase activities are appropriately mapped to those SyRS requirements assigned to software. If all risk control measures have been represented by SyRS requirements, then verification of software (SRS) requirements need only look to the SyRS for risk control design input. If the risk management activity has created its own document or database of risks and control measures that have not been traced to system-level (SyRS) requirements, then verification of SRS requirements for completeness may necessitate trace linkages to those risk management documents or databases.
2. The task above is retrospective in the sense that it is addressing risk control measures that were defined in the past. There is also a corresponding prospective task. Once the software requirements are complete, or nearly complete, much more is known about the anticipated operation of the software, and thus, the device itself. It is appropriate at this time to conduct another risk management review to identify new risks or new hazards that result in previously anticipated risks. Having more details about the function of the software allows more detailed analysis about how that software might

fail in ways that could result in a hazard. In short, the analysis needs to be repeated given the new level of information available. It may be that new risk controls are needed for newly identified risks or hazards. Perhaps previously identified risk controls will now found to be ineffective based on the way the software requirements have developed. This new analysis will likely lead to additions and changes to the risk management's control measures that will need to be reflected in the SRS, or perhaps the SyRS or other specifications in which control measures need to be detailed as requirements.

## Who Should Write the Software Requirements?

It seems that those who complain the loudest that detailed requirements will restrict their creativity often do not want to participate in the development of the software requirements. They do not seem to realize that the development of requirements is the most creative part of the development life cycle. This is the time to creatively address the system identified in the SyRS in ways that will meet the needs of those who will use or otherwise interface with the device when it is completed.

Requirements development is best done “in the sunshine.” That means that the requirements should be created with maximum visibility to all those who will be affected by them. That list is a long one that ranges from users and clinicians through the manufacturing and service personnel who will work with the device daily for years. This list of “stakeholders” may or may not be involved in actually writing the requirements, but they must be involved in reviewing the requirements before they are implemented. Who better to determine if the software, as documented in the software requirements, will meet stakeholder needs than the stakeholders themselves? When better to determine the acceptability of the software concept represented in the requirements than before the time and resources are expended in designing, implementing, testing and debugging the software?

This suggests a nice orderly progression of writing, reviewing, modifying, and accepting requirements before they are designed and implemented—not who should write them. Requirements can be written by anyone in the organization who is able to understand and balance potentially conflicting stakeholder needs. The author(s) must be able to create and evaluate potential solutions to those needs and craft a proposed solution that is acceptable to all stakeholders. This task requires the ability to organize one’s thoughts related to the solution in a way that is easily communicated to those who ultimately will review those requirements. Finally, the author(s) must have the ability to design those requirements in a clear, precise, unambiguous way that ultimately is verifiable when the implementation is complete.

Writing requirements is not an easy job. When the range of interests and skills of the potential audience for those requirements is considered, it is almost an impossible job! It definitely demands a skill set of its own. So, the best answer to the question of who should write them is a dedicated team of requirements developers who do nothing but author requirements and referee the ensuing debate and negotiations

that take place as they are being reviewed and modified. Oh, yes, the requirements author also needs to be part diplomat!

Not all manufacturers have a product development team big enough or have enough new product development in the pipeline to have dedicated requirements specialists. So, what are the next best options? Really, anyone with an interest in learning this as a new skill, who has good communication skills, and who has an interest in designing how the product will work can do this job. Job title makes little difference.

Often, those charged with validation are charged with requirements development. That's because they are also charged with verifying that the software requirements are an accurate expansion of the SyRS requirements. They are also one of the audiences that will use the requirements most extensively as they later develop their system-level software test protocols. They have a vested interest in the end result being concise, unambiguous, and testable.

Software developers, too, have a vested interest in having concise, unambiguous, implementable requirements and should make good requirements authors. That is, assuming they intend to use them as their guidelines for the design and implementation of the software. Unfortunately, that is not always the case. Some software developers choose not to write requirements or ignore them if they are written by others. Admittedly, it's a lot more fun to develop software by trial and error. The near instantaneous stimulus-response cycle one gets from writing a little software and trying it a minute later is, indeed, addictive. However, the objective of creating medical device software is not to maximize the entertainment value of the job for the developer. The objective is to create safe software that fulfills the needs of the device stakeholders. To do that successfully without requirements that have been agreed upon by the stakeholders takes almost incredible good fortune.

A project team that has invested hundreds of hours in documenting stakeholder needs, distilling them into a negotiated approved SyRS, will not appreciate creativity that creeps in at the design or implementation phase of the project that ignores those needs. The best way to insure that they are represented in the end product is to design through software requirements for how the software will function and interface with the user; review the requirements for consistency with SyRS intent, then design and implement the software to fulfill those software requirements.

Creativity at the design and implementation phase is not a good thing, at least creativity in *what* the software is to accomplish. Appropriate design phase creativity focuses on *how* the software is constructed to implement what the requirements say it should do. Developers who want to provide more creative input in determining what the software does should do so. However, they should do so by getting engaged early in the development of the software requirements—not by single-handedly taking this on at the keyboard hidden away in a cubicle.

Take for example a device manufacturer who invests months of time, and tens or hundreds of thousands of dollars in human factors research that results in well-defined user interface needs and requirements. Fonts, sizes, colors, icons, button sizes, screen placements, even the wording of critical messages were determined to best fit the needs of the intended users in the intended use environment. Whether this was embedded into an SyRS or software requirements specification (SRS) is immaterial, because, in our example, the company's software developer took it

upon himself to deviate from almost all the visual user interface requirements (if indeed he was aware of them) because of his personal preference, or because he found something easier to implement, or his new software tool set came with a nice default set of fonts, buttons, and icons that he liked. Whatever the motivation, his implementation ultimately had little to do with the very expensive recommendations of the human factors experts. In his mind it was better, but not in the minds of the intended users. His creativity, while satisfying to him, had little value, and in fact, cost his employer dearly in time and expense to resolve the software that creatively did not meet the software requirements.

Does that mean that the developers aren't capable of better creative ideas, or should stifle them if they do have them? Of course not, it simply means that those creative recommendations need to be aired in the sunshine just as all the other requirements were before they are implemented. To repeat an earlier point, engaging software developers in the requirements development process during requirements phase activities is much more efficient than waiting to get their input on requirements in the later design and implementation phases, and much, much better than finding out by surprise during final test or in the field what the developer has implemented in seclusion.

## The Great Debate: What Exactly Is a Requirement?

I'm sure I've spent many man-weeks of my life going back-and-forth with employees, clients, and hecklers about what is requirement, what is design, and what should and should not be documented in requirements. I wish I could say it does not matter, but it does. I wish I could say it's very clear, and there is a standard, or checklist to help make the determination, but it's not always clear, and for every checklist there are always exceptions.

Let's start with the "is it a requirement or is it design" question. A set of system requirements defines what the device will be. A set of software requirements defines what the software must do to meet certain system requirements by detailing the behavior of the system as viewed by the user and controlled by the software. The design specifies how the software will be structured to meet the intent of the software requirement. So, it seems that the requirements are "whats" and the designs are "hows." That sounds pretty easy and to a large extent it is—at least until we start thinking about it a little (too much).

Part of the problem is one of perspective. One reader's requirement is another reader's design. Suppose a software requirements specification (SRS) includes requirements for a graphics user interface (GUI). Further, suppose those requirements include detailed requirements for how the screens transition and what events cause the GUI to transition from screen to screen. To give a better high-level understanding of the GUI operation, the requirements are summarized in the SRS as a flowchart. That flowchart gets a reaction from software designers who complain that this is too much design detail. Their opinion is that this should be their territory, and they should have latitude to design the GUI screen interactions as they see fit to meet the real requirements—of which this is not.

The clinical and human factors specialists, however, don't see it that way. They are aware of some context confusion with competitive devices GUIs. They have invested a lot of time with intended users and consultants working through the GUI operation to come up with this screen flow to minimize menu depth and make the interface more intuitive. They even tested rapid prototypes with the users to optimize the screen flow and to document user satisfaction with the design. They claim this level of detail is, by all means, a requirement. The last thing they want is a design and implementation to come back that takes a radically different approach. That makes it a requirement, right?

Does that mean a developer couldn't come up with a better idea that intended users would like better? No. It only means that if they do come up with alternative ideas, they need to be subjected to the same level of scrutiny as those proposed "requirements" to see if, in fact, the intended users (and consultants, human factors experts, marketing, clinical experts, etc.) agree that the new idea is better. The developers complain that this will take a long time. That's right, it does, and that's why everyone else is eager *not* to make changes. The developers need to be more engaged in the requirements process while it is taking place to avoid time-consuming changes later on.

Context also makes a difference. The very same sentence may be a requirement in one context, and a design detail in another. For example, suppose a laptop-based user interface for a bench-top medical device is described in an SRS. The very first requirement is that the software must operate on a Windows platform. "Foul," cry the developers. Certainly the choice of operating system is a design-level detail. This clearly is getting into how the device is designed, not what the device is to do. Besides, a month earlier, when reviewing the SRS for the robotic front end of the very same device, everyone agreed that the choice of embedded operating system was a design-level detail. What gives?

This time the marketing and clinical folks step forward to point out that the device will be in a lab surrounded by other Windows-based instruments. Introducing an operating system that is perceived by the user as being different leads to additional training, could lead to mistakes being made, and may prove to be a competitive disadvantage against the chief competitor who is operating his or her user interface on the Windows platform.

In this circumstance, the choice of operating system may well be considered a requirement (or a design constraint).

Of course the verification and validation test team wants to weigh in on the requirement vs. design debate, too. Their view is colored slightly differently based on what they see as their needs for requirements and designs. Their opinion is likely to be determined by asking themselves, "Should this be tested at the system level?" If the answer is "Yes", then they are likely to decide that it is a requirement. If the answer is "No" or "It can't be" then maybe it is design-level detail that is better tested in integration- or unit-level testing. The tester's world is compartmentalized into which type of test would be most appropriate for the requirement/test in question. To the extent that testers like to organize their system-level tests to trace to SRS, and their unit and integration tests to link to design descriptions and interface descriptions, their opinions on this debate are more black and white, or at least suffer from fewer shades of grey.

What happens if requirements do end up in a design document? That often means that there probably is not enough detail in the SRS for comprehensive system-level testing. The result may be that testers may not dig through an SDD looking for system-level requirements. Important functionality may not be tested. Scattering requirements across a large number of design documents makes the tester's job nearly impossible simply because they don't know where to look for requirements. Further, testers treat requirements like requirements regardless of where they find them, thus restricting that freedom the developers thought they had. Requirements may be found in an SDD, or verbally in a review meeting, at the water cooler, over the cubicle wall, or (worst of all) by observing how the device operates once it is implemented.

What if design elements end up in the requirements document? That does restrict the designers and implementers. Sometimes that is the intent as in the operating system selection example above. When it is intended it is a good thing. However, inappropriate design detail at the requirements level becomes a constraint that sometimes puts the designer in the position of deciding whether to implement something that is thought to be wrong (or not as good as it could be) going to the trouble of requesting a requirement change, or just ignoring the requirement.

## Anatomy of a Requirement

First and foremost, a good requirement is a simple requirement. Writing requirements simply is complicated and difficult. Counterintuitively, it seems harder to write simpler requirements. However, the likelihood that a requirement will be understood and acted upon is much greater when the requirement is simple.

A good, simple requirement is made up of three components: the prequalifying circumstances, the action, and the modifiers of the action. For example:

When the pressure in the patient line exceeds 200 mm Hg for more than 500 contiguous milliseconds, the pump shall stop within 100 milliseconds seconds.

The prequalifying circumstance here is the line pressure exceeding a limit for a set period of time. The action is that the pump must stop, and the modifier of that action is that it must stop in 0.1 seconds or less.

If only life were always that simple. In this example, it is likely that the prequalifying circumstance (over pressure) should also be accompanied by an alarm, and a lock-out preventing the user from restarting the pump with the high pressure present. The temptation is to put all three actions into one big requirement. For example (i.e., a bad example):

When the pressure in the patient line exceeds 200 mm Hg for more than 500 contiguous milliseconds, the pump shall stop within 100 milliseconds seconds, the overpressure alarm shall be asserted until acknowledged by the user, and the user shall be prevented from restarting the pump by any means until the pressure is reduced below 200 mm Hg.

This is a compound requirement, and should be avoided because:

1. Simply put, it is run-on, ambiguous, and difficult to understand.
2. Using the trace as a metric for completion is confounded by the fact that having a single trace to this compound requirement could lead us to the conclusion that the requirement is fulfilled when it is really only part of the requirement that has been met.
3. Traceability to test procedures is likely to be confusing since each of the actions may take several test procedures to cover. This compound requirement is likely to have many test procedures tracing back to it, and coverage will be difficult to ascertain.

How might this compound requirement be better organized? Consider the following:

1. Overpressure condition: When the pressure in the patient line exceeds 200 mm Hg (overpressure threshold) for more than 500 contiguous milliseconds:
  - 1.1 The pump shall stop within 100 milliseconds seconds;
  - 1.2 The overpressure alarm shall be asserted until a silencing event occurs;
  - 1.3 The user shall be prevented from restarting the pump by any means until the overpressure alarm is cleared.

This organization assumes the same prequalifying circumstance in 1, without having to copy that circumstance to each requirement related to it thus creating a document maintenance problem.

Let's dig a little deeper. Alarm and alert handling is often complicated. Let's further expand 1.2:

- 1.2 The overpressure alarm shall be *asserted* as a *Class 3 alarm* until a *silencing* event occurs;
- 1.2.2 When the ACK membrane switch is pressed, the alarm shall be *acknowledged*;
- 1.2.3 When the pressure drops below the overpressure threshold continuously for more than 1 second, the alarm condition shall be *cleared*.

Accept for now that this is a very simplified, hypothetical example. Much more could be said, and the organization could be improved. However, notice the words in the above that are *italicized*. Recall the discussion about building a language to support validation from Chapter 1. This is a good example of what was discussed at that time. Assuming that the terminology (i.e., language) for alarm functionality has been clearly defined elsewhere in the requirements, all this makes perfect sense. It would be clear what asserting, silencing, acknowledging, and clearing mean. In fact, there are likely to be sets of requirements where those terms are defined that clearly state what requirements are to be met for “asserting a Class 3 alarm,” or for “silencing an alarm by acknowledging it.” This implies a hierarchical organization of the requirements beyond the simple outlined organization of the requirements shown above.

In fact, these requirements start to look like source code! Assuming we want all Class 3 alarms to behave identically, there is little value in repeating the functional

requirements for a Class 3 alarm for each event that triggers one. Software developers will see it that way too, and will most likely design one instance of the software for handling all Class 3 alarms. By formally developing a language in the requirements, it is possible to speak in a shorthand language unique to the requirements document. Not only does this shorten the requirements document, but it resolves ambiguity, reduces maintenance effort (e.g., if someone decides to change how Class 3 alarms behave), and implies how the underlying code might be structured.

It is not necessary to pursue this example any further. The points have been adequately made. Requirements generally are composed of three components (prequalifying condition, action, and the modifier of the action). Some of those components may be shared among a number of requirements, and the requirements can be structured hierarchically to reduce repetitiveness and improve maintainability. Additionally, requirements can benefit from formal definitions of terms and grouping of requirements that clarify the terms.

Notice the use of the word “shall” in the example above. There is a group that gets up-in-arms over the form of the verb “to be” that is used in requirements. The imperative form “shall” seems to have been settled on by the U.S. defense industry some time ago as the verb that clearly identifies a sentence as a requirement. Certainly, there is a large following out there that demands a “shall” or they ignore the requirement. The actual choice of the verb really isn’t that important, but it is important to choose one and use it consistently. Remember that requirements work demands precision in the use of language. Consider the following examples that all are found in the same software requirements specification:

1. When the pressure exceeds 100 mm Hg the device *shall* be in an overpressure state.
2. In the overpressure state, the OP alarm *should* sound.
3. The OP alarm *may* sound until the pressure drops below 100 mm Hg.
4. In the overpressure state, the pump *will* stop.

For now, focus on the selection of the verb in each of the requirements. Admittedly, there are other ways these sample requirements could be improved. The inconsistency of the use of the verb in each requirement adds to the ambiguity in the specification. Assume for our example that all four of these are meant as software requirements. The following ambiguities are possible after using “shall” in requirement 1 above:

2. In the overpressure state, the OP alarm *should* sound. Is this a requirement for the software here, or is this commentary that the alarm *should* sound as a result of a requirement elsewhere? Perhaps it *should* sound, but under some circumstances it *may* not?
3. The OP alarm *may* sound until the pressure drops below 100 mm Hg. Does this mean it *may*, but it’s equally acceptable if it doesn’t? Perhaps it means that the alarm *may* sound for as long as pressure is above 100, but it’s OK if it doesn’t sound that long. In other words, does this mean the alarm *may* sound as long the implementer chooses—as long as it doesn’t sound after the pressure drops below 100?

4. In the overpressure state, the pump *will* stop. Is this the software's responsibility to stop the pump, or is this commentary that the pump *will* stop by other means (e.g., a hardware implemented shut down)?

As one who may be in the position of verifying or implementing these "requirements," you will need to know if each of these is requirement or commentary and what exactly the requirement means.

Now, on your own, reread the four original requirements, substituting the word "shall" for the italicized verb in each requirement. Knowing that all requirements are noted as such by the use of the verb "shall" makes it clear that each is a requirement, not commentary. Furthermore, "shall" is more definite than verbs like "may" that leave it unclear if the sentence is a requirement at all, and if so, whether it is intentionally flexible or not. What would a verification test look like for "the alarm may sound"? If it sounds, it passes. But if it doesn't sound, wouldn't it also pass?

Even softer language is sometimes seen in requirements specifications using verbs such as:

- Could;
- Recommend (that);
- Request (that);
- Desire;
- Would like;
- Might.

This kind of soft language may be acceptable in user needs documentation that defines what users have expressed as goals for the design. However, in a lower-level requirements specification, use of words like these makes the document read like a stream-of-consciousness monologue of design alternatives. They don't provide hard guidance for the implementers, and leave the verification test designers clueless about how to develop a test for something that "might" happen. Language like this in a requirements specification may be indicative of indecision, or a paucity of information available to make a firm requirement. "Soft requirements" using verbs like these run a high likelihood of not being implemented, or of being implemented in a way never anticipated by the author of the requirement or the test designer. Someone once wrote that one could substitute "probably won't" for any soft verb used in a requirements specification. "The alarm should sound" becomes "The alarm probably won't sound." If that substitution isn't acceptable, then maybe the requirement should be "hardened" a bit (I don't remember where I read this to properly credit the originator of this idea.)

There are those who intentionally use softer verbs to indicate the difference between a "hard" requirement that cannot be ignored and a "soft" requirement that is more akin to a goal than the requirement. In these types of requirements specifications, the use of these verbs is explicitly detailed early in the document. There is nothing wrong with this approach, but those who are not accustomed to being very careful with their choice of words often lapse inadvertently into using the incorrect form.

If a requirements specification will contain requirements of varying “hardness,” perhaps a better way of specifying the degree of hardness is to use a requirements tool that allows the addition of an attribute field for each requirement. That attribute could be used to grade how “hard” the requirement is in any way desired, even on numeric scale.

One thing is certain in developing requirements: The words do matter!

## How Good Requirements Are Written

It has been said before and it will be said again: writing good requirements is difficult. A good requirements writer needs to possess some domain knowledge about the device for which the requirements are being written. Further, the writer must have some level of technical competence in both the science behind the device and the engineering skills needed to design and implement the device. The writer must be a good interviewer, listener, psychologist, mind reader, diplomat, organizer, designer, inventor, manager—and a good writer.

The requirements phase, as said before, is the most crucial phase for both the device developers as well as those charged with validating the end result. Let us spend a little time looking at where good software requirements come from, the process of collecting them, and what a good requirement actually looks like.

Unfortunately, writing requirements is not as simple as closing oneself in an office for weeks and writing them in seclusion. It takes more than simply polling or interviewing intended users of the device and organizing their thoughts into a requirements specification. Indeed, one will find that intended users seldom know exactly what they are looking for in a device. This is where the interviewing, listening, and mind reading comes into play. There is a whole discovery process involved with requirements development that takes creative questioning, proposed wording, and interpretation of the results received from intended users. There is a fine line between creatively extracting requirements from the device stakeholders and proposing solutions. Too often requirements writers jump ahead to propose solutions without really digging down to understand the root of the requirement—the need. This does often push requirements close to or over the line of becoming designs, and this is part of why there is a debate about requirement versus design.

Those who have written requirements for any length of time are well aware of the fact that software requirements frequently change. Sometimes they change simply because of market conditions. For example, if another company comes out with a competing device that contains the new feature, it is a safe bet that your marketing group will want something similar regardless of what phase of your software development life cycle you were in. Difficulties in design and implementation may be encountered, and the team may find that it is easier (or even better) to change the software requirements to make them easier to implement. Gaps, inaccuracies, or inconsistencies may be discovered as the implementation progresses. All of these will necessitate requirements changes. The requirements development and management process must be flexible enough to deal with the inevitable change to the requirements that will take place throughout the life cycle.

Closely related to change is the fact that there is no such thing as a perfect requirements specification. There is always room for improvement. More requirements can be added, or more detail provided. Design detail can be replaced with more abstract requirements, inaccuracies can be corrected, and so forth. If the life cycle process waits for requirements to be completed before initiating design, projects will seemingly (or actually) take forever to get into design and implementation. The best requirements development processes are those that recognize the iterative nature of requirements design. There is no reason to think that requirements design can be done linearly when we openly admit that software design and implementation needs to be done iteratively.

Earlier, the structure of a good software requirement was discussed. Let us spend some time on the contents of good software requirements, so that we can write them, recognize them in reviews, improve them, or just understand why we may have trouble implementing or testing them.

### Clarity and Accuracy

First and foremost, a requirement must be clear; that is, easily understood and not ambiguous. Clarity is difficult to achieve without some level of independent review. Further, the best reviews for attacking ambiguity are review meetings, not electronic circulation of documents for review. Why is that? Sometimes ambiguities are easily recognized as such. The reader himself or herself realizes that a requirement could be interpreted as meaning more than one thing. However, almost as often, ambiguities are missed because the reader of the requirement thinks that he understands the requirement, assuming that his understanding is the only understanding of the requirement. The next reader may similarly think that he or she understands the requirement, but that understanding may be totally different from the understanding of the first reader. And so it goes, each reader in a serial review circulation of a requirements specification could have a totally different understanding of what the requirement means, and yet none of them would flag the requirement as being ambiguous because they thought that they had the one and only understanding of what the requirement intended. This is why the role of the reader is so important in the review process for requirements. If, in a review meeting, the reader is asked to *interpret* a requirement in his or her own words, it is more likely that others in the review meeting listening to this interpretation will come to recognize their difference in understanding.

There are things we can do to promote clarity and accuracy during the requirements development process. Those include:

**Avoiding compound requirements.** Compound requirements are almost always ambiguous. The number of permutations and combinations of the conditions in the compound requirement are seldom all covered in a compound requirement, leaving the reader to interpolate the intent of the requirement in all conditions.

For example, consider the following compound requirement:

The device shall begin processing the samples when all three interlock sensors are inactive, and shall initiate an alarm if they are active.

We superficially understand what is intended, but we do not *exactly* understand what was intended. Can we assume that processing does not begin if any one or more interlock sensor is active? Is the alarm only initiated when the user attempts to begin processing the samples, or is it initiated as soon as the interlock sensor goes active? Is an alarm initiated for any number and combination of interlock sensors that are active? Is there only one generic alarm type for any combination of active interlock sensors? Does sample processing start automatically once the active interlock is cleared? Once started, does sample processing stop if an interlock sensor goes active?

Each component of this compound requirement had some ambiguity in it. The fact that we have two ambiguous requirements compounded into one sentence will make clarification in the requirements specification much more difficult than if we were simply dealing with two separate ambiguous requirements.

*Avoiding exceptions in requirements.* Exceptions usually include words like: except, unless, although, but, when, and if. They usually make for bad requirements because they create difficulty for testers in creating situations in which the requirement is *not* fulfilled. For example:

When the pressure exceeds the *high pressure threshold*, the *overpressure alarm* shall be *asserted unless* the user has *disabled* alarms.

Note that requirements with exceptions are closely related to compound requirements. In the preceding example, the first part of the requirement could stand alone as a requirement for assertion of an alarm when the initiating event has occurred. The clause that follows “unless” is actually a second requirement for silencing alarms when disabled by the user. This exception would have to be added to every requirement for alarm assertion when written this way. If it were not repeated for every alarm assertion requirement, it would be ambiguous whether disabling alarms applied to each condition or not.

*Avoiding synonyms.* Writing requirements can sometimes get a little monotonous. Some requirements developers are tempted to experiment with language to make reading the requirements more entertaining, or at least more interesting to write. Again, this leads to ambiguity. In the pressure alarm example given above, note that certain words are italicized. The writer of these requirements has developed a specific language that is used consistently throughout the requirements specification. Terms that are italicized are specifically defined or are assigned quantitative values in a single location within the specification. Now, imagine that the corresponding requirement is required for an underpressure condition. The overpressure requirement is repeated below (without the exception clause) followed by the new underpressure requirement:

When the pressure exceeds the high pressure threshold, the overpressure alarm shall be asserted.

When the pressure sensor indicates a pressure below the low pressure threshold, the underpressure alarm shall sound.

In this example, the italics for the special requirements language have been removed. The second requirement is very similar to the first requirement except that it addresses low pressure instead of high pressure, and a few of the words have been changed. Although the changes are fairly minor, and either requirement could stand more or less on its own, the fact that the terminology has changed between requirements introduces ambiguity. As a reviewer, one would have to question where or how the overpressure is being detected if only the low pressure is being detected from the pressure sensor. Further, the overpressure alarm is asserted, but the underpressure alarm is sounded. Does that mean that there are differences between the two alarm types?

One should avoid the temptation to try to make requirements interesting or entertaining. They are not intended to be entertaining; they are working documents. In a best case scenario, a language or set of definitions is defined early in the specification, used consistently throughout, and somehow made to stand out as a special term. This not only makes the meaning of the requirements less ambiguous, it creates a common language that can be used by multiple contributors to the requirement specification.

*Avoiding negative requirements.* These are shortcuts that we all are tempted to use at some time or another. Negative requirements also lead to ambiguity. Consider this requirement:

The pump speed cannot be calibrated by the user.

First of all, this is not really a requirement because it can be met by doing nothing. Furthermore, how would one test this? Would the test procedure attempt to calibrate through a number of means, and, when unsuccessful, claim that the test passed? That is not a very productive way to spend expensive engineering time, is it? Perhaps the requirement means that the user should be unable to calibrate the system, but that a service technician can. The fact that we started the previous sentence with “perhaps” indicates we are guessing what the requirement means, and therefore there is an ambiguity. However, if that was the intent, the requirement have been better worded as:

When the authorized service password is entered, the service mode option screen shall be displayed.

The service mode option screen shall provide an option to calibrate the pump speed.

By converting the negative requirements into two positive requirements, we have clarified the ambiguity related to who exactly is able to calibrate the pump speed (users with the service password). It also made it clear exactly how to get to that calibration functionality, and was stated in two testable, positively stated requirements.

*Traceable.* Requirements should be written in a way that will facilitate traceability to the design requirements that will take the requirements a step closer to imple-

mentation, and to the verification tests that will ultimately verify correct implementation of the requirement.

There are several things that can be done to write requirements in a way that promotes good traceability. First and foremost is a good organization of the requirements specification. This is obviously important for other reasons as well, but one might consider when organizing the requirements how the design document is likely to be organized, or how the test protocols are likely to be grouped. Traceability is much easier to implement and much easier to understand if the source and destination documents are following similar organizational structures.

Although we have already covered this, compound requirements lead to complex traces. This is one more reason to avoid compound requirements.

Why are easily traceable requirements important? The development and validation test teams are more likely to get real value out of trace links if they are maintained, easily understood, and relatively simple to keep up to date. Few tasks are less satisfying than trying to update trace links to two documents that are several revision cycles out of sync. A good trace from requirements to verification test(s) will make review of the verification test much simpler by making it clear that the test or tests linked to a specific requirement represents a complete and thorough test of the requirement. This helps avoid gaps in testing as well as over testing the same functionality by unnecessary retesting.

*Testable.* The best advice for how to make a requirement testable is to think about what the test would look like for the requirement at the time the requirement is written or reviewed. As mentioned above, negative requirements are impossible to test, compound requirements are difficult to test, ambiguous requirements will give ambiguous results. Beyond avoiding those pitfalls, using precise language and verifiable criteria wherever possible help to make a requirement more testable.

Quantitative values in requirements make them more testable as do tolerances on certain performance specifications. For example, consider the following requirement:

The medication shall be delivered at a rate of 1 mL per minute.

It is unlikely that any test using any precise means of measurement will provide a result of exactly 1 mL per minute. So, if the device delivers the medication at 0.99 mL per minute or 1.01 mL per minute, is that acceptable? If those are acceptable, then at what point would they be unacceptable? Perhaps a slight underdelivery is acceptable, but an overdelivery is never acceptable. One would never get that from the requirement the way it is currently written. A better way of writing this requirement would be:

The medication shall be delivered at a rate greater than or equal to 0.95 mL per minute and less than or equal to 1.00 mL per minute.

*Readable.* Yes, everything matters when it comes to requirements. For them not to be ambiguous, they need to be readable. To promote good practices related to reliance on the requirements, they need to be written in a way that they are easily

used and do not become a bottleneck in the design, development, or validation process. Some of the attributes discussed above such as using terminology consistently, and explicit definitions of that terminology are best practices for making requirement specifications more readable.

There are other very simple techniques for making specifications more readable. In fact, just keeping the requirements simple by avoiding run-on sentences and by using common words will achieve much to make the requirements more readable. Keeping the vocabulary limited to a minimum number of readily understood words will make it readable to the largest possible audience, especially if that audience includes readers whose primary language is not the same as that of the author(s). Likewise, using simple direct sentences maximizes readability for all the same reasons. Using a consistently uniform structure for all requirements, while not the most entertaining to read, makes each requirement uniformly easy to understand. This also reduces reader fatigue that can result from trying to read between the lines to understand why one requirement is written one way, and a similar requirement written a totally different way.

Writing requirements for complex logic can be especially challenging. Even worse, sometimes the more one tries to explain complex logic, the more complex the explanation gets and the lower the likelihood that the reader will understand. Although traceability to a flowchart, state diagram, or other graphic representation of complex logic is problematic, the graphic representations are often much easier to understand than pages of requirements detailing the logic. There is no reason that a requirements specification cannot include graphics that are treated more like narrative or comments in the requirements specification. To facilitate meaningful traceability, individual requirements represented by the graphic should be broken down into traceable language in the specification. However, oftentimes, the graphic itself suggests the organization for the requirements that will describe it, and when those requirements are accompanied by the graphic (even as narrative), they become more understandable.

For example, consider the software requirements for a stopwatch whose state machine is shown in Figure 11.2. (Granted this isn't a medical device, but it is simpler than most medical devices for the purpose of this example.) The stopwatch has a single display for elapsed time, and two user inputs: the Start/Stop button and the Reset button.

Our template for the requirements to describe the operation of the stopwatch is:

State Name

Entry Event (the events that brought us into this state)

Functional Requirements (the requirements while in this state)

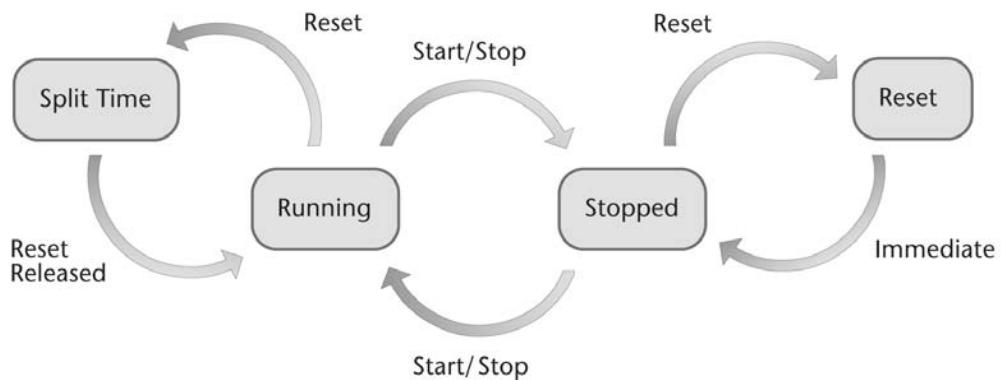
Exit Event (the events that take us out of this state)

Simple, right? Let's see how it works for the simple state machine of Figure 11.2.

## 1. Running State

### 1.1 Entry Events

- 1.1.1 When the START/STOP button is pressed from the STOPPED state.



**Figure 11.2** Example of a state machine for a stopwatch.

1.1.2 When the RESET button is released while in the SPLIT TIME state.

### 1.2 Functional Requirements

1.2.1 The internal timer shall increment continuously in 0.01 second increments.

1.2.2 The display shall show the contents of the internal timer at all times.

### 1.3 Exit Events

1.3.1 When the START/STOP button is pressed, the watch shall enter the STOPPED state.

1.3.2 When the RESET button is pressed, the watch shall enter the SPLIT TIME state.

## 2. STOPPED State

### 2.1 Entry Events

2.1.1 When the START/STOP button is pressed from the RUNNING state.

2.1.2 Immediately after the RESET state completes its functions.

### 2.2 Functional Requirements

2.2.1 The internal timer shall stop within 0.01 seconds.

2.2.2 The display shall show the contents of the internal timer at the time it was stopped.

### 2.3 Exit Events

2.3.1 When the RESET button is pressed, the watch shall enter the RESET state.

2.3.2 When the START/STOP button is pressed, the watch shall enter the RUNNING state.

## 3. RESET State

### 3.1 Entry Events

3.1.1 When the RESET button is pressed from the STOPPED state.

### 3.2 Functional Requirements

3.2.1 The internal timer shall be reset to zero.

3.2.2 Button presses shall be ignored while in this state.

### 3.3 Exit Events

- 3.3.1 The watch shall enter the STOPPED state immediately after the functional requirements for the state are complete.

## 4. SPLIT TIME State

### 4.1 Entry Events

- 4.1.1 When the RESET button is pressed from the RUNNING state.

### 4.2 Functional Requirements

- 4.2.1 The display shall stop at the time shown when the state was entered.
- 4.2.2 The internal timer shall continue to increment uninterrupted.
- 4.2.3 START/STOP button presses shall be ignored while in this state.

### 4.3 Exit Events

- 4.3.1 When the RESET button is released, the watch shall enter the RUNNING state.

That's it for the user interface/operational requirements for our software-implemented stopwatch. There certainly will be other requirements related to the display, accuracy, and precision, but the state machine that describes the operation is captured here. Further, it is structured in a form that could easily be copied by the design and implementation. Note the simplephrases and sentences, consistency of structure, and special terms in capital letters.

Before moving on, let's think for a few minutes about the entry events in the example above. You may have noticed that they are not full sentences and are not written like other requirements. Each one of them could have been completed with the implied "the XXX state is entered." There really is no good reason not to complete the sentences for the sake of consistency. However, there is another issue. Each entry event is a duplicate of an exit event from another state. In fact, one could argue that only the exit event of the entry/exit pair is truly a requirement. The entry events are really only passive requirements.

So why do we duplicate them? There are two good reasons for this. First, it makes each state self-contained, which is important for using these requirements as a working document. Without the duplication of the entry events, one would have to thumb through the requirements for all the other states to figure out how the state machine ever got into a given state. Second, it is a good cross check of the completeness of the requirements. Whenever we have used this technique, as part of the review process we go back to check that the duplicate transition requirement actually exists in the requirements. It is interesting how many times we find errors using this method. Any trick we find that helps us find errors/defects in this early requirements phase is a trick we will continue to use!

Another way our sample template could be improved for more complex examples might be to include the status of fixed user inputs (such as buttons) and outputs (indicators or displays). For example, certain buttons may be active and others ignored in a given state as they are in the SPLIT TIME state of our example. Similarly, certain indicators may be on, off, or flashing in different states of the device. These two additional template items were not used in the example because of its simplicity. The same requirements were handled in the functional requirements template item. One can readily appreciate that if we had 5 or 10 buttons, and half a

dozen displays and indicators that the additional organization offered by these two fields would be welcomed.

## Summary

As with other chapters, we have only scratched the surface on this topic. There are several outstanding books dedicated to this topic. Two that come to mind are the Karl Wiegers [3, 4] texts.

In wrapping up this chapter, it bears repeating one last time that, in my opinion, the development, review, and management of software requirements are the most important activities of the software development life cycle, whatever that life cycle may be. It is rare, if ever, that I have observed a successful development project for a device containing software that did not have strong requirements. By successful, I mean not only that the resulting device was safe, effective, and a success in the marketplace, but also that the project ran under some semblance of control, and finished reasonably close to expected costs and timeframes.

Skeptics might ask just how often software projects finish close to projections. The answer is “not often enough.” However, one reason for that is that not enough device manufacturers are taking the requirements seriously. It is puzzling why the medical device industry struggles with requirements when the NIST report suggests that software developers across industries have increased their focus on requirements—and it’s not a regulation for most of them.

Good requirements habits are the key to better medical devices. Further, their development practices and adherence to those practices are primary targets for improvement for companies who want to reign in development and validation costs and schedules.

## References

- [1] Tassey, G., “The Economic Impacts of Inadequate Infrastructure for Software Testing,” May 2002. Available at <http://www.nist.gov/director/prog-ofc/report02-3.pdf>.
- [2] Andersson, M., and J. Bergstrand, “Formalizing Use Cases with Message Sequence Charts,” unpublished Master’s thesis, Lund Institute of Technology, Lund, Sweden, 1995.
- [3] Wiegers, K. E., *Software Requirements, 2nd edition*, Microsoft Press, 2003.
- [4] Wiegers, K. E., *More About Software Requirements*, Microsoft Press, 2006.



# The Design and Implementation Phase Activities

## Introduction

If you are a reader who is looking to this chapter for established or recommended ways to design and implement software, you will be disappointed. If you are hoping that the regulatory requirements and guidelines will specify what languages to use, and how to structure software in a standard way, you similarly will be let down.

This textbook has rarely prescribed a single recommended approach to any validation activity, and will not start when it comes to design and implementation. Many in the industry wish that the regulatory agencies would be prescriptive in some of the validation methodologies and outputs. “Just give me a form with the right questions and I’ll fill it out,” is heard many times each month from our clients. There are reasons that regulators (and this text) do not offer specific forms and prescriptive methods. In fact, developers for medical device manufacturers probably would not want or use prescriptive validation and development templates and forms if they were available ... even though they clamor for them now! This is because there is not a single methodology or single documentation style, format, or list of contents that would be equally applicable for the entire industry, for every project, forever. Standardization at too high a level will only lead to frustration as most projects struggle to fit design solutions into forms that are not a good fit.

Most likely, developers probably are relieved that their technical designs are not constrained by regulatory requirements. Neither the FDA nor any other regulatory agency is going to specify what computer language should or should not be used. There is no regulatory information available about indentation rules, naming conventions or other details that would be found in coding standards (although it *is* recommended that manufacturers have coding standards). There are no rules about what development platforms or embedded operating systems should or should not be used. There are no recommended or forbidden software architectures, design documentation styles, development tools, or anything else that would constrain the developer from exercising his or her creative juices to write software—that implements the software requirements as specified.

In Chapter 5, as part of a discussion of development life cycles, agile methods of software development were mentioned. The lean documentation artifacts and the rapid iterations of software that are hallmarks of agile would give the appearance of being software whose design is not controlled, at least by GPSV standards. Remember that the GPSV is a guidance document, not regulation. It is quite possible within the bounds of the design control regulation that an alternative agile version of GPSV could be created that meets regulatory requirements, even though it is at odds with

the GPSV. As of the writing of this text, AAMI has convened a workgroup of industry and regulatory representatives to author a technical information report (TIR) to attempt to address some of the issues that would reconcile agile methods with design control regulation.

This chapter will focus on design validation activities other than testing activities. Recognize that in reality, there may well be significant activity in this phase readying system-level (black box) software verification tests based on the software requirements specification. Actual sequencing of tasks is specified in plans to fit an overall life cycle model. That has been covered in detail in earlier chapters. The testing activities will be covered in the next chapter. It is left to the reader to understand how this all fits together chronologically for the specific circumstances of the project at hand—and each one is likely to be different from the last.

## Regulatory Background

There are no regulatory requirements that are specific to design activities. The closest is the QSR regulation 21 CFR 820.30 (d) on Design Output:

(d) Design output. Each manufacturer shall establish and maintain procedures for defining and documenting design output in terms that allow an adequate evaluation of conformance to design input requirements. Design output procedures shall contain or make reference to acceptance criteria and shall ensure that those design outputs that are essential for the proper functioning of the device are identified. Design output shall be documented, reviewed, and approved before release. The approval, including the date and signature of the individual(s) approving the output, shall be documented."

—*Quality System Regulations: 21 CFR 820.30 (d)*

As regulatory requirements go, this is about as generic as they come. What is specific is the requirement for “documented, reviewed, and approved” design output documents, which certainly would include any documentation of the design activities that are covered in this chapter. Equally specific is the requirement for documented approvals of such design outputs including the signature and date of the approval.

Why might a regulatory body be so specific about reviews, approvals, and dated signatures on design outputs, yet so generic about the requirements for what actually is contained in those outputs?

The reason for being generic is as discussed earlier in this chapter. Regulators do not want to limit device manufacturers to specific content or format which may, in fact, not fit every conceivable medical device. Additionally, too much detail in regulation would stifle creative, and perhaps better, approaches to design in the future.

The reason for the specific regulatory attention to reviews and approvals with dated signatures has to do with the intent of the design control regulations of the QSRs. That intent is, simply, that the design process is controlled. Do approvals with dated signatures control the design process? They do if the organization fol-

lows a process that requires signed and dated approvals before moving forward in the life cycle.

Signatures and dates are important to regulatory agencies because they leave an auditable trail for establishing the level of compliance with the manufacturer's quality system processes, which presumably are compliant with regulatory requirements. Of specific importance are the dates of the signature. The sequencing of documents according to the approval/signature dates establishes whether the activities in the design control process were followed in the expected order. That is, of course, if the approvals were not fraudulently backdated. For example, if the final validation test report is signed and dated six weeks before the software requirements specification is signed and dated, an auditor might suspect that the software requirements were simply reverse engineered from the software, and that little if any control of the design process existed.

The GPSV is a bit more specific about its expectations for validation activities related to the design process. To paraphrase, the GPSV suggests:

- Software design evaluations;
- Traceability analysis;
- Analysis of communication links;
- Reexamination of the risk analysis;
- A formal design review.

Interestingly, the GPSV once again reinforces that there is no expectation that the design activities take place only once during the development life cycle. In fact, the following quotation from the GPSV seems to openly invite partial internal release and iterative design:

Portions of the design can be approved and released incrementally for implementation; but care should be taken that interactions and communication links among various elements are properly reviewed, analyzed, and controlled....

Most software development life cycle models will be iterative. This is likely to result in several versions of both the software requirements specification and the software design specification. All approved versions should be archived and controlled in accordance with established configuration management and documentation control procedures.

The activities itemized above are the validation activities for the design phase of the life cycle. The development tasks are not specifically itemized in GPSV, but certainly include development of the software design specification (SDS). Although the GPSV does not specifically mention the creation of the SDS, it does itemize recommendations for the content of that document which support software validation:

The software design specification should include:

- Software requirements specification, including predetermined criteria for acceptance of the software;
- Software risk analysis;

- Development procedures and coding guidelines (or other programming procedures);
- Systems documentation (e.g., a narrative or a context diagram) that describes the systems context in which the program is intended to function, including the relationship of hardware, software, and the physical environment;
- Hardware to be used;
- Parameters to be measured or recorded;
- Logical structure (including control logic) and logical processing steps (e.g., algorithms); Data structures and data flow diagrams;
- Definitions of variables (control and data) and description of where they are used;
- Error, alarm, and warning messages;
- Supporting software (e.g., operating systems, drivers, other application software);
- Communication links (links among internal modules of the software, links with the supporting software, links with the hardware, and links with the user);
- Security measures (both physical and logical security); and
- Any additional constraints not identified in the above elements."

—*General Principles of Software Validation: Section 5.2.3*

The guidance goes on to say that the first four of these bullets usually exist as references to other documents external to the SDS. Note that some of these bullets are fairly specific, and others are less so. However, one thing is quite clear, the expectation of the GPSV is that the SDS is more than a simple block diagram, or a half-hearted theory of operation description of what the software does.

## Validation Tasks Related to Design Activities

Although the creation and maintenance of the software design specification (SDS) should be the responsibility of the development team, some understanding of what goes into an SDS is needed by the validation team to review and evaluate designs and design traceability in a meaningful way. For that reason, this section begins with a look at the SDS.

### The Software Design Specification (Alias the Software Design Description)

The software design specification (SDS) is also frequently called the software design description (SDD). The two terms are used interchangeably in this text.

As mentioned previously, the task of developing the SDS is generally the responsibility of the development team. The SDS is the document (or set of documents) in which the software requirements of the SRS (or equivalent set of documents) are translated into a technical design description of the software that should be used as a guide to implement those requirements. It is often said that the requirements are the “whats” and the designs are the “hows.” The requirements specify *what* the software is doing (from the perspective of the user), and the designs specify the organization, logic, calculations, data structures, communications protocols, and all other technical details of *how* the software will be written.

Why is an SDS or other design documentation needed? There are several good reasons that they add considerable value to the design, development, and validation processes. The arguments for design documentation can be divided into prospective and retrospective categories. Of course, if design documentation is not completed before implementation is undertaken, then the prospective value of the documentation is greatly diminished or totally lost. This is the most common reason for the complaint that design documentation is a waste of time. It is, if it is merely an exercise in documenting what has been created by ad hoc design methods.

Prospectively, design documentation provides an opportunity to elaborate on the intended designs so that they can be reviewed for their completeness, accuracy, and consistency in fulfilling the software requirements. Additionally, design details can be reviewed for their impact on safety. Software hazards and failure modes that would not have been predictable without design-level detail can now be analyzed, and control measures can be contemplated to reduce the risks associated with such hazards. Further, all of the advantages of traceability that are covered later in this chapter are not possible unless there are design-level details available that can be traced back to the software requirements. Finally, the feasibility of implementing the design and the feasibility of testing the design can be assessed during review, but only for design-level details that exist before the software is implemented and the tests have been created.

Ultimately, at some point in the life cycle the software does have to be implemented. If the design details are sparse or totally missing, the software engineer is likely to make ad hoc decisions about what goes into the software at the time of implementation. This leads to “invisible” software that is not likely to be tested independently, and may not perform in a way that would be acceptable to the majority of the stakeholders having an interest in the device.

Retrospectively, the main value of design documentation is for the long-term maintenance of the software. If repairs or feature upgrades are to be made to the software in the future, there is a high likelihood that those who will be responsible for those changes may not be the same ones who originally created the software. If they are the same, there is a high likelihood they will not remember every detail of the design of the software. The design documentation should provide both the high-level and detailed-level design information that will be useful to someone who may have to pick up the responsibility for the software in the future.

What goes into the SDS or other design documentation? In the regulatory background section of this chapter, Section 5.2.3 of the GPSV was referenced for the FDA’s guidance on the content of an SDS. There is no need to repeat that detail here. Instead, let us consider how that detail might be organized in a way that promotes both the prospective and retrospective uses of design documentation.

As with many other topics discussed in this book, there is no one solution that will work for all device software. The IEEE does publish a recommended practice [1] for design documentation that provides some excellent guidance and recommendations on content and organization. The needs of a multiprocessor-based highly complex system are much different from those of a simple single microprocessor application. In general terms, what all software design documentation has in common is a need for low-level design detail (LLD) that is put into context by a

high-level description (which is often called the high-level architecture or HLA) of how the different functional blocks of the software fit together.

One can easily recognize that highly complex multiprocessor designs may have several levels of HLA that, at the top level, clarify how each microprocessor or microcontroller in the system is related to each other, and what communication interfaces exist among them. Each microprocessor's software itself deserves a high-level description or HLA of the software modules of which it is comprised. Often the HLAs are comprised of block diagrams for the major blocks of software functionality. Each interface to other major blocks of software, whether through electronic or other types of communications, data structures, or user interface, is noted on the HLA diagram. Complex interfaces that are common for electronic communication interfaces often require LLD documents to describe the communications protocols right down to the level of message content, sequencing, timing, error handling, error recovery, constraints, and assumptions upon which the communications protocol is based.

Each block and interface link on an HLA of diagram is usually deserving of some low-level description somewhere. There is great value in LLD design documentation that goes down to the individual function (or object) level that details:

- Inputs (including units where appropriate);
- Outputs (including units where appropriate);
- External data structures referenced;
- Internal data structures maintained;
- Algorithms or other logic descriptions;
- Assumptions and dependencies.

Documenting to that level of detail provides one with more than enough detail for peer evaluations and broad-based design reviews of the design. As with requirement reviews, it is impressive just how much debugging actually can be accomplished during this design and review process. Defects, in fact many defects, can be identified and resolved before any time is invested in implementing the code. By the time one implements the designs, several rounds of this debugging has already been completed, and one finds (at least in our experience) that implementation is much more straightforward and takes place in much less time than if the software engineer had to invent on-the-fly in a random walk fashion. Furthermore, this level of documentation makes it easy for one to add software engineers (really, programmers, even junior programmers) during implementation to shorten schedules. This is facilitated by the design documentation because the system-level considerations have already been taken into account during the documentation and review process. This means that the new recruits do not need overall system-level visibility to write the source code to implement the function design for which they are given responsibility. If, for some reason they do need higher level visibility, it's already documented.

Although this level of detail is quite valuable, placing this level of detail in a document that is separate from the source code creates a maintenance task to keep the SDS document in synchrony with the source code itself. This "document thrashing" is not a prized assignment among software engineers. In fact, once the software source code has been implemented, the value of maintaining the SDS low-level detail

is of questionable value except for the high level retrospective maintenance purposes. This is because software engineers claim almost universally that for maintenance (retrospective) purposes they would almost always refer to the source code before they would refer to SDS low-level detail.

As an alternative, one might place and maintain low-level detail in the source code itself in the form of headers for each function or object. This eliminates the need to keep the two design outputs in sync, and puts explanatory low-level detail in the source code itself, which is where software engineers are most likely to look for it in the future. Of course, there is still a need to assure that a process of designing and reviewing before implementing is followed. The temptation to skip design/review and jump straight to source code development is a powerful one.

## Evaluations and Design Reviews

Chapter 9 has covered the topic of reviews. That discussion distinguished between peer reviews, evaluations, and formal reviews. In the GPSV's discussion of design activities, it mentions evaluations and reviews separately. Assuming that reviews are meant for the less technical reviews of the design activity outputs to inform management, the software design evaluations should be (as the GPSV says) "conducted to determine if the design is complete, correct, consistent, unambiguous, feasible, and maintainable." To that I would add "testable." We will not delve into evaluations and design reviews anymore here. The topic has been dealt with adequately in Chapter 9. However, it is worth noting here that one role the validation team can play in a medical device development project is to verify that the software designs meet those attributes specified in the GPSV, as well as other attributes that may contribute to the success of the project.

## Communication Links

Why would the GPSV go out of its way to single out communication links as an area that deserves special attention during the design activities? First of all, what do they mean by that? Any link between major subsystems, whether those subsystems are separate microprocessors running separate pieces of software or software subsystems that communicate through any means, represents a communications link. Communications might be wired or wireless, through shared memory structures, even button presses or other user inputs. So, why are communication links deserving of special attention? Communication links are areas for which there is a high likelihood of error. It is quite possible, even likely, that different design groups have designed the software on either side of a communication link (if, indeed the link requires software on both sides). It is inevitable that certain assumptions are made by both groups as they design and develop their software. If the two groups make different assumptions, there is the potential for an error.

Consider the Mars Climate Observer (MCO) of 1999. After it crashed into the Martian surface, an investigation was launched, which concluded:

The root cause of the error was units—metric versus English. MCO was designed using metric units. However, the Mars Climate Observer (MCO) Mishap Investigation Board found that the “Small Forces” software file was coded to send output

data in English units. The spacecraft was sending telemetry data (in metric) and a trajectory estimate (in metric) back to Earth, both of which were relayed through the English-based Small Forces file [2].

In other words, the spacecraft was calculating and communicating in metric units; however, another software item, which interfaced with the spacecraft assumed English units. The altitude above the surface of Mars was calculated correctly. However, the results were interpreted incorrectly because of the incorrect assumption of units. The flawed interpretation resulted in an incorrect understanding of the MCO's trajectory and altitude. Forgive my layman's interpretation of the consequences if not interpreted correctly. The point remains that communication links are a high likelihood source of defects in systems. Unfortunately, this example resulted in the destruction of the spacecraft and a total loss of a mere \$327.6 million project. Oops.

Since communication links are higher probability areas for discovering defects, they deserve special attention in review during the design phase. In fact, one could argue that communication links deserve their own requirements/design specifications, as well as specialized communication link testing. They tend to be more complex than they appear to be. That makes them prime candidates for "under-thinking" that comes with confidence and results in defects. Communication links are shared by the two (or more) end points of the communication. Often the responsibility for design and test of communication links is ambiguous, resulting with both sides assuming (or just hoping) the other is taking responsibility for testing.

## Traceability Analysis

Why would the GSPV recommend traceability analysis as one of the design phase activities of the software development life cycle? Recall that validation is required to be sure we have designed the *right product*. By that we mean assuring that user needs (and other stakeholder needs) are accurately translated into requirements, designs, and ultimately into the software itself. One way to accomplish this is to trace from user need to requirement, and from requirement to design. Additionally, traceability provides objective evidence that a controlled design process was, in fact, in place when the product was developed.

What is meant by controlled design process in this context? As related to design activity, some evidence that all of the requirements are represented in the design, and that there are no elements of the design that are not called for by the requirements. Specifically, what one needs from traceability during the design activities is traceability from software requirements to design requirements, and vice versa. Additionally, if it has been decided not to funnel risk control measures through system requirements and/or software requirements, then they must at least be represented in the design requirements.

Figure 12.1 shows what traceability from software requirements to design requirements might look like. In this figure, software requirement number three does not trace to any design requirement. This is a sign that the software requirement will be lost in implementation. Likewise, design requirement E is a design requirement that is not called for by software requirement. This kind of orphan design requirement could mean one of two things. Either the designer is designing in

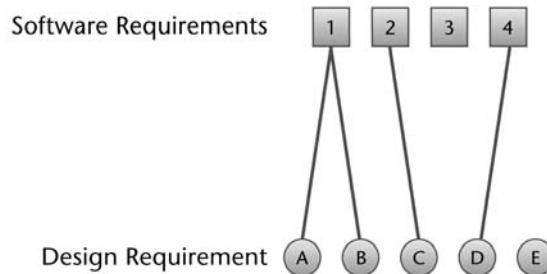


Figure 12.1 Requirements to design trace.

functionality that is not actually needed, or the designer has come across a need for the design requirement that was not anticipated by those who wrote the software requirements. The danger of leaving a design requirement unlinked to any software requirement is that it reduces the likelihood that that element of the design and resulting implementation may be missed by system-level testing activities. Orphaned design elements should be carefully assessed for whether the corresponding software requirement should be created in the SRS.

That is why traceability analysis is important. Now, what does it look like? Anyone who has tried this realizes that it is not an easy or satisfying task. Anyone who has tried to retro-document a project probably has struggled with the traceability analysis and felt it was a waste of time. This is another one of those situations in which it probably was a waste of time. (However, to be clear, that is not an excuse for not completing a regulatory suggestion or requirement.) The true value of traceability analysis is during the design process, not weeks or months after the fact. Further, the traceability analysis will in large part depend upon how the software design is documented. The level of detail and organization of that documentation determines the ease of producing the trace and value of the resulting analysis.

Consider the need for traceability analysis from software requirements to design descriptions to the source code itself. The simplest and most valuable advice that can be offered here is that the more similar in structure and organization these three design outputs are, the simpler and more valuable the trace analysis will be. If the software requirements are left for team members with no software development experience, there is little likelihood that the SRS will be structured with any semblance to the SDS or source code structure.

Let us consider the following simple, the common examples. Suppose a diagnostic device has requirements to create three different printed hard copies of the diagnostic test results. The requirements document has three sets of requirements for the three reports. Each set of requirements contains individual requirements related to the formatting of the report and the contents of the report. The requirements are reviewed and approved by the users, and handed over to the design team for design and implementation.

The designers set about documenting their software design for creating these reports, and realize that there is more software to be designed to simply output a single character to the printer than there is at a higher level to create the content and format specified in the software requirements. For instance, as shown in Figure 12.2, the requirements shown at the top of the figure are adequately represented in

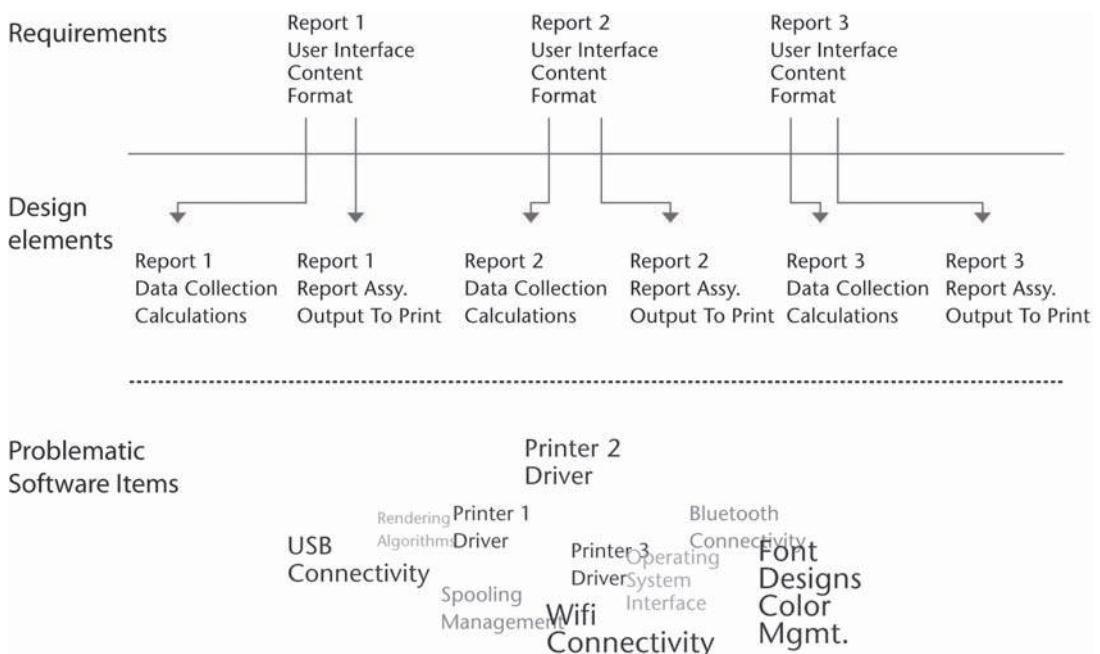
the design in the top layer of design requirements above the dashed line. What is left is all of the software below the dashed line. The software below the dashed line has no clear direct trace back to a specific requirement or set of requirements.

Because of the way the requirements are written in this example, all of the software design requirements below the dashed line are implicit, assumed, and largely invisible to anyone but the designers and implementers of the software. Anyone interested in testing that software is going to have to find it by reading source code, figure out why it exists, understand how it should work—then develop a test for it. It's not likely that this will be pursued for full coverage of the software. Could it be any more complicated and costly to test the software?

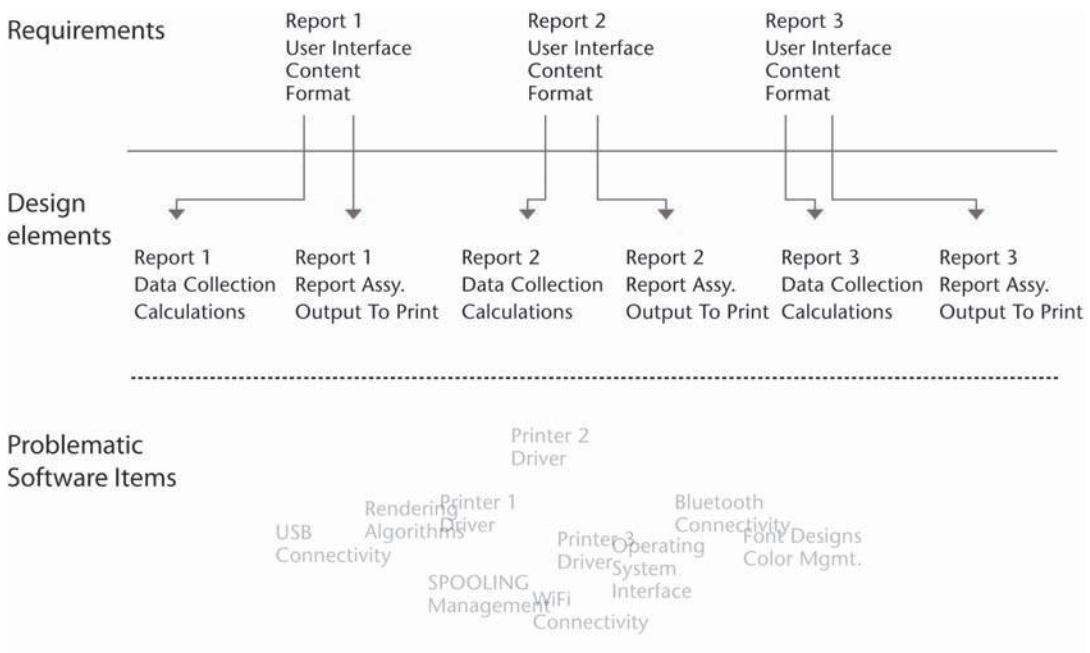
Referring again to Figure 12.2, one might ask how all that software below the dashed line might be traceable to the three requirements at the top of the page. Two possibilities that have been observed numerous times, and are not recommended are represented in Figures 12.3 and 12.4.

Figure 12.3 represents the “head-in-the-sand” approach (not really a technical term) to design traceability. The designer works with the requirements he or she has, traces to the design elements that specifically address those requirements, and although he or she must be aware of all the software below the dashed line, it is simply ignored as far as traceability analysis is concerned, thus making it “disappear.”

Figure 12.4 represents the “spaghetti” approach (again not a technical term, but is descriptive) to design traceability. In this case, the designer recognizes all of the software design elements below the dashed line, but satisfies the traceability task by tracing the three software requirements to everything below the solid line. That may



**Figure 12.2** Printed report example of software requirements and design requirements.

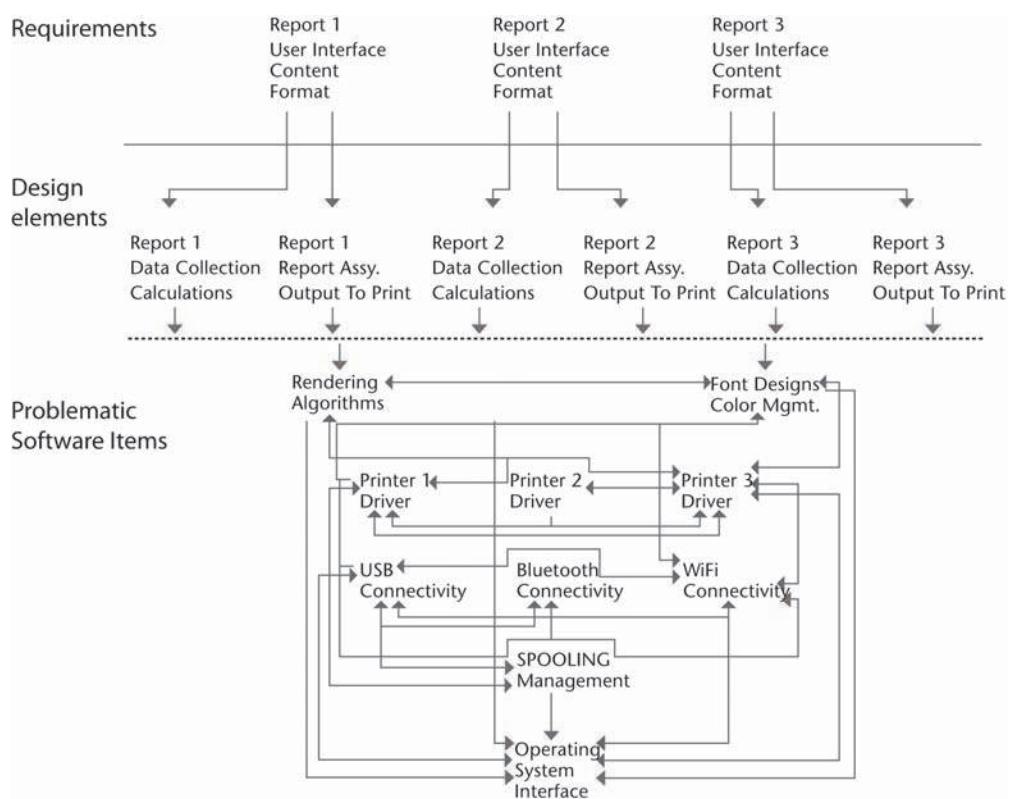


**Figure 12.3** Head-in-the-sand traceability. Troublesome software simply disappears.

satisfy the GSPV's recommendation for traceability, but it is hard to see how any effective analysis could be done from this.

So, what kind of value would we like to create from traceability analysis? Suppose, in the above example, that for some good reason printer spooling (that is, buffered background printing) is not a desired feature for this device. Without some traceability to and from the requirements, designers and/or developers might reasonably assume that printer spooling is an implicit requirement. Not until after the design is complete and possibly implemented would anyone notice that the printer spooling feature were in place. There is a possibility that even after implementation it would not be noticed. It is reasonable that the testers implicitly assumed that printer spooling was not in place (or never thought about it, or weren't aware of what printer spooling is) and therefore never designed tests that would exercise print spooling or tests to confirm that it is not present. The head-in-the-sand approach would never document or trace down to a level that would show the print spooling. The spaghetti approach may well have documented it, and even traced to it, but because everything is obscured by far too many trace links, it is likely that few would take the time to do a thorough analysis.

What might work better? The basic problem with the above example is simply that more software needs to be implemented than is detailed in the software requirements. Should the requirements authors have been able to anticipate all this detail? If they were technically savvy with software, perhaps they could have anticipated more than the three requirements. However, once the designers realized that they had a lot of software to design that did not trace back to requirements, they could have, and probably should have gone back for changes to the requirements to support the reality of the situation.



**Figure 12.4** Spaghetti traceability.

Figure 12.5 shows an alternative arrangement that could result from such an iterative approach. Additional requirements were added to detail the specific types of printers that this device would support including what connectivity would be supported to each of the printers individually. Adding the additional detail to the requirements cleans up the traceability to the design with the ultimate benefit of making it easier to analyze. For example, careful analysis of this trace diagram shows that while all three printers can be connected through a USB connection, only Printer 1 connects via Bluetooth and only Printer 3 has a WiFi connection. Further, Printers 1 and 3 do not require special rendering algorithm software while Printer 2 does.

Why might that be important? Perhaps the system requirements intent was that all printers would only be connected through USB connectivity. After all, extending connections to WiFi and Bluetooth opens up the issue that more requirements will be needed for configuration of these connections.

Issues related to print spooling may, in fact, be issues that the requirements stakeholders have strong opinions about. Requirements related to printer spooling may raise issues related to hardware support (may require more memory) or related to patient privacy (HIPAA regulations related to encryption or the potential of long-term data storage). With either the head-in-the-sand or the spaghetti traces, it is unlikely that these issues would have been surfaced at the design review stage. The

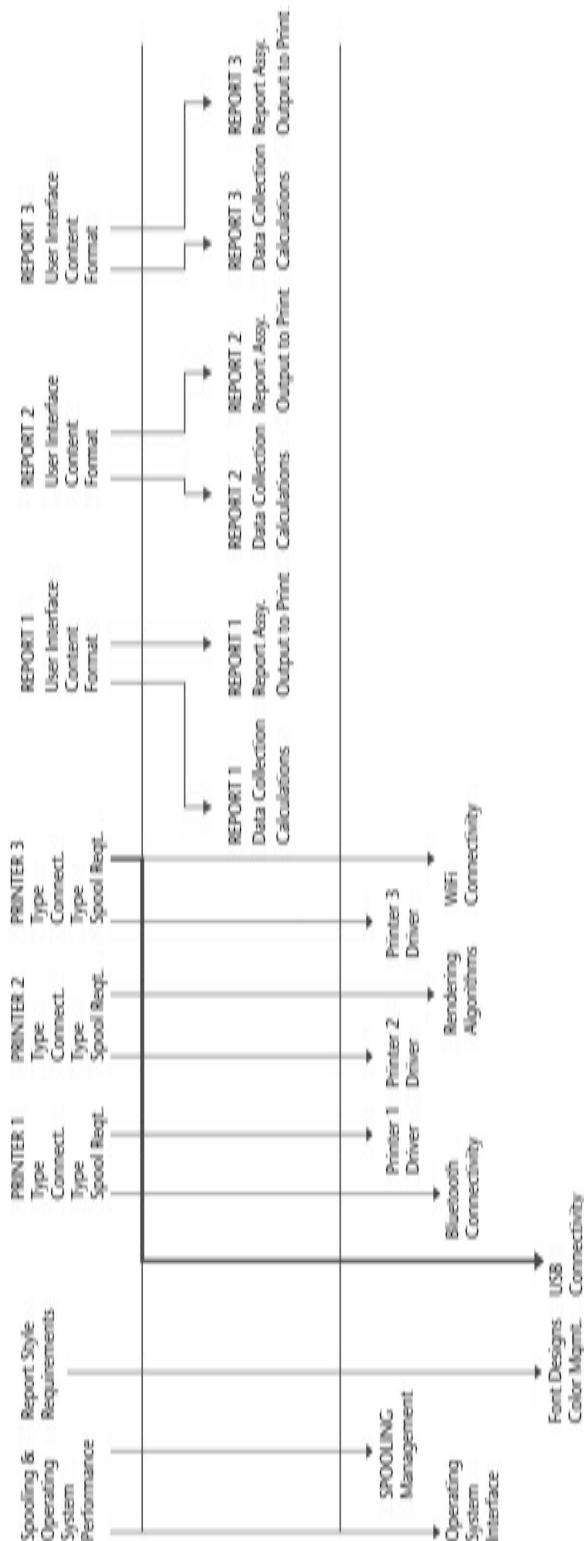


Figure 12.5 An alternative design trace schema.

traceability method called attention to the ambiguity in the intent, raised issues for discussion and review, and ultimately documented the result in a traceable manner that is more easily understood by reviewers, inspectors, and those tasked with maintaining the device design in the future.

There is one last point to make related to traceability, and that is the question of resolution of traceability to design requirements. One might reasonably expect that a software requirement may require dozens of design requirements to fulfill it. Is there value in tracing a single software requirement to dozens of individual design elements? Probably not, unless those design elements are scattered across many different sections of the SDS, and ultimately across many different source files in the software itself. If that is the case, that level of detail in the trace may be necessary to find all the pieces of the design and implementation. However, a simpler, more understandable approach is to organize the design documentation in such a way that related software requirements are organized together in the SRS, and likewise, related design elements that will implement those clustered requirements also are organized together in the SDS. If that is the case, then traceability can be handled by tracing *sections* of the SRS to *sections* of the SDS, much like the example in Figure 12.5 shows. Is this cutting corners? No, as long as it is not abused by tracing hundreds of software requirements to thousands of design requirements in one trace link, it actually will improve reviewers' ability to understand how the requirements are being implemented in the design.

Without covering it separately, all of the considerations above related to the organization and resolution of traceability from requirements to design are also true for traceability from design to source code. Anything that can be done to organize the source code in the same way that the design document is organized will make the traceability almost implicit and require very little additional documentation.

## Risk Management

In Chapter 8, the point was made several times that risk management is not a one-time event in the software life cycle. Since risk management is considered to be a validation activity, let us briefly run through the risk management activities that should be considered in connection with the design activities of the software development life cycle.

By the time the actual design is underway, a number of the risk control measures (alias mitigations) identified in previous iterations of risk management should have made their way into requirements, and now into the design itself. Part of the risk management activity at this point in the life cycle is to review those risk control measures to be sure that (a) they are indeed implemented, and (b) that they are implemented adequately to address the risk which they were intended to control.

As the design is created and documented, a new opportunity for risk management and hazard analysis arises. At this point in the life cycle those individual software modules and off-the-shelf (OTS) software items that will make up the device software can be identified. Risk analysis can be conducted for this new level of detail. Failure mode and effects analysis (FMEA) is now possible since there are individual software items that are identifiable, and the failure modes of those items can

be analyzed. In short, at this point in the software development life cycle, one needs to assess whether failure modes of the software design and embedded OTS software could lead to hazards that were not previously identified. Additionally, one needs to consider whether the design created new failure modes that were not previously anticipated. Ultimately one needs to consider whether these hazards could lead to new harms and therefore new risks, or whether these hazards could impact previously identified risks by altering the severity of the resultant harm, or by impacting the likelihood of those hazards resulting in harm.

Oftentimes, some of the human interface details are left to the designers and implementers. If this is the case, one needs to look at these new designs to assess hazards and resulting risks related to human factors issues created or addressed by the software design and/or implementation. Simple examples of this include:

- Checking any new messages to the user for clarity (are the prompts, messages, warnings, and alarm text free of ambiguities that could be misinterpreted in unsafe ways?).
- Intuitiveness of the user interface (can the user figure out how to navigate the user interface without training or having to reference the operator's manual?).
- Performance of the user interface (can the user navigate it fast enough in potentially safety critical situations?).

The relatively short treatment of risk management here is not intended to imply that it is not very important among the design activities. It is very important. The treatment here is short simply because the topic was covered at length in Chapter 8.

## Validation Tasks Related to Implementation Activities

Obviously the main activity among the implementation activities is the creation of the source code itself. Some of the validation activities are very similar to those discussed above for the design activities, namely:

- Compliance with detailed design specs; that is, verification that the implemented code is an accurate and complete representation of the design as described in the SDS or other design documentation;
- Traceability; that is, verifying that all of the design requirements of the SDS or other design documentation can be traced to the software, and that all of the software is in some form represented by the design documentation.

No more will be said on these two tasks simply because they are so similar to the tasks for the design activity. There are, however, several validation tasks that are specific to the implementation phase that do need to be covered in some detail:

- Coding standards and guidelines;
- Checking reuse of preexisting software components;
- Documentation of compiler outputs;
- Static analysis.

## Coding Standards and Guidelines

Coding standards are standards or guidelines developed by the device manufacturer to standardize certain levels of detail, style, and quality in the source code that is implemented by the developers. Generally speaking, there is a different coding standard for each language that the developer uses for implementing the device software. These standards include conventions agreed upon among the developers related to the naming of variables and constants, tabulation, naming of functions or objects, size and complexity guidelines for individual functions or objects, and guidelines related to in-line comments and headers.

These standards tend to be highly individualized from company to company and subject to the opinions and tastes of the software developers. However, one can easily find coding standards on the Internet that can be used as a foundation for creating a customized version. There are so many available, there is little value in referencing any of them specifically. Using any Internet search engine, one needs only to look for such terms as “C coding standards” and “C# coding standards.” There are also tools available for automatically checking source code for compliance with coding standards.

Why should your developers have and use coding standards? There are several reasons, and developers who work with coding standards generally become great believers in their value. Guidelines for suggested complexity levels increase reliability and maintainability of software, and make it easier for testing at the unit test level. Complexity levels can be measured using a number of complexity metrics, and there are tools available for automatically and objectively calculating complexity. Guidelines for naming and commenting make the software easier to understand and more maintainable by others in the future. Even simple things like standardizing on tabulation and other elements of style somehow make it easier for one engineer to maintain another engineer’s software. More advanced coding standards go beyond the superficial and standardize on specific coding constructs. The details of these standards are beyond the scope of this text. However, one of the validation tasks during implementation should be to ensure that coding standards have been adopted by the organization, and that they are being used consistently during the implementation of the software. Retro-engineering source code at the end of a project to meet coding standards not only adds little value to the process, but runs a high risk of injecting errors into code that may have already been tested and working properly.

## Reuse of Preexisting Software Components

One of the results of the investigation into the Therac disaster was that “dangerous reuse” of software components was partially at fault for the disaster. As device manufacturers mature in their use of software to implement functionality in medical devices, they reach a point at which they recognize that certain functionality is unchanged from device to device. When that functionality has been implemented in software, it seems only natural to reuse those software components on a different device, thus leveraging the investment in development of the software. There is really nothing wrong with doing this, and in some cases it may even be more reliable to use preexisting software with some field experience than to write it from scratch.

However, there is a danger in being lulled into a false sense of confidence simply because the software has been in use previously.

There is a temptation to lightly test, or not test at all software functionality that is carried over from another device. The danger in that logic is that the “intended use” of that software functionality may be slightly different from one device to the next. For example, in the Therac incident, a critical function implemented in software on a prior generation of the device worked safely, but when reused on the Therac 25 the same software produced catastrophic results. The earlier generation of software had relied upon the assumption of a hardware safety mechanism that was implemented differently on the Therac 25. Once the assumed safety mechanism was removed, the software that had been perhaps defective but safe in the past now became defective and unsafe.

When reusing software, its state of validation is cannot be separated from the system of which it is a part. In other words, if a function works properly in one system, one cannot assume that it will work properly in a second and different system. Those responsible for validation (and design) of systems that contain reused software must be aware of any implicit or explicit assumptions upon which the reused software was based. Further, it is not valid logic to skip testing critical, safety dependent software simply because it is being reused from another device and has no history of problems.

The above discussion on reused software was focused on proprietary software that is part of another device belonging to the device manufacturer. However, the same logic would apply to off-the-shelf software or software library functions. One cannot assume that they are safe and problem-free simply because they have been used in hundreds or thousands of other applications. They are still deserving of analysis and management for their contributions to software-related hazards. Validation test plans should include testing of off-the-shelf components necessary to build confidence that safety related failures do not exist.

### **Documentation of Compiler Outputs**

This may seem like an odd validation task, but there is some evidence of regulatory expectations of this in Section 5.2.4 of the GPSV. It recommends that:

... at the end of the coding and debugging process, the most rigorous level of error checking is normally used to document what compilation errors still remain in the software. If the most rigorous level of error checking is not used for final translation of the source code, then justification for use of the less rigorous translation error checking should be documented. Also, for the final compilation, there should be documentation of the compilation process and its outcome, including any warnings or other messages from the compiler and their resolution, or justification for the decision to leave issues unresolved.

Anyone who has written software is aware that compilers are capable of outputting streams of warnings and alerts about the code it compiles. These generally are of warnings of nonstandard usages, suspected errors, and so forth. Although most are benign and may have been seen hundreds of times during development, some may be valid alerts of errors in the code or of errors in the way it will be inte-

grated with other code. Additionally, because the warnings have been seen many times, the developer may suffer from warning fatigue and overlook a new and serious warning simply because he or she is accustomed to seeing a stream of warning messages.

The activity recommended in the GPSV is likely to be sparse in its defect detection ability. That is, dozens of benign warnings may need to be justified for every real defect found. There is a risk in making late changes to source code for the sole purpose of reducing the compiler error and warning count to be justified. There is a likelihood that errors will be injected into the code in the process. One should never alter the source code to correct a compiler warning or error without thoroughly retesting the code that was changed. The validation team needs to be alert for the need to document compiler outputs and should verify the completion of the task. Perhaps more importantly, the change controls and configuration management must be up to the task of catching any late changes to the software that are made to reduce compiler messages late in the development life cycle.

Why would a device manufacturer want to take on a task that has a relatively low likelihood of finding real device software errors, and has a real risk of injecting errors in response to benign warnings? Quite simply, because a device manufacturer would not want to be in a position of explaining how device defects found their way to the patient with errors that were caught by the compiler.

## Static Analysis

Detailed code inspections are very good at detecting software problems by inspection without running the executable code. This is sometimes referred to as static analysis of software. Automated software static analysis tools exist that check for more complex errors than are found by most compilers. These error types are related to syntax and even common run-time errors and poor programming practice (such as redundant code and dead code). The more complex static analysis tools are capable of detecting casting errors, memory leaks, race conditions, buffer overruns, and other common errors found in software.

How do static analysis tools fit into a validation plan? More analysis is better than less, but the most sophisticated static analysis tools are very pricey. Without question, low-cost tools like LINT provide a lot of good information at a reasonable cost.

As of the date of this writing there is no regulatory mandate or even guidance related to static analysis tools. However the FDA's CDRH Office of Science and Engineering Laboratories (OSEL) has formed a Software Forensics Lab for testing static analysis tools on various manufacturers' device software—and have been known to request or require justification for all of the findings of the tool.

The best recommendation at this point in time is to watch for any news from OSEL that may indicate a move toward recommending or requiring the use of static analysis tools. Companies that can afford the advanced tools probably will benefit from their use. Companies that cannot afford the tools should definitely make use of the more affordable tools, and document a detailed code inspection process that probably will be more effective than the advanced static analysis tools anyway (in my opinion).

## References

- [1] IEEE 1016-1998: IEEE Recommended Practice for Software Design Descriptions.
- [2] [http://askacademy.nasa.gov/mars\\_climate\\_orbiter\\_goes\\_silent](http://askacademy.nasa.gov/mars_climate_orbiter_goes_silent).



# The Testing Phase Activities

## Introduction

At last. It is time to cover testing software. When you started this book, you may have thought that it was a book on testing methods. That may mean that you were one of the many who equate “testing” with “validation.” Hopefully, the message has been delivered that “validation” includes much more than “testing,” although testing is a very important component of validation.

As with every other chapter, there is much to cover. The treatment of testing here is very broad, covering nontechnical topics like the psychology of software testing and test management. Additionally, slightly more technical coverage is given on test types, testing methods, test metrics, and test tools. Recognize that all of these major topics could be, and are, subjects of their own books. The treatment of testing here is not to replace these other texts, some of which will be referenced below. Instead, the objective is to give the reader a broad appreciation of the topic as it applies to the medical device industry.

By grouping the discussion of the testing activities in one chapter, it is not meant to imply that testing is only done in a single compressed phase or time frame of the software development life cycle. In fact, preliminary testing during the implementation of the software is probably the most productive activity for finding defects in the software. An entire chapter was devoted to the validation life cycle in Chapter 7. That should have made it clear that the testing tasks (designing, developing, executing) are staggered throughout the design, implementation, and final test phases of the software development and/or validation life cycle.

## Regulatory Background

As we have seen in other chapters, the regulations themselves all refer to “validation.” Regulatory suggestions or recommendations related to software validation are clarified in General Principles of Software Validation (GPSV), which has been referenced many times in earlier chapters. Sections 5.2.5 and 5.2.6 of the GPSV deal specifically with testing. In contrast to other parts of the GPSV, these sections get rather specific in their details. Rather than pull all that detail into this section only to cover it in later sections, this chapter will only deal with some generic comments in the GPSV in this section. Some of the more detailed guidance will be covered in the appropriate section later in this chapter.

At times, when involved in a validation project for a life-critical device, it feels like all of the responsibility for the safe and correct operation of the device is on the shoulders of the validation team. This is especially true for projects in which the

development team is less than cooperative. Certainly, the testers are always the last to finish the project (you cannot test it until *after* it is developed). Therefore, to some extent the testers really are the last hope of finding any defects in the device before it is put in the hands of the user. That is an awesome responsibility considering the level of complexity of some devices, the demanding schedules, and the lack of appreciation that exists in some organizations for finding the dozens of defects that ultimately assure the safety of the device, but will keep the device from meeting its release schedule. To be sure, the testers almost always are the bearers of bad news. If it is any comfort, at least the FDA appreciates how difficult the job is:

It is a time consuming, difficult, and imperfect activity. As such, it requires early planning in order to be effective and efficient.

*—General Principles of Software Validation, Section 5.2.5*

So there it is; testing is time-consuming and difficult. It is also imperfect. We can never test every defect out of the device. If there is an expectation that the testers should be perfect in their craft, then why not place the same expectation on the developers? Fortunately, there are things that can be done to control the efficiency, effectiveness, and complexity of the testing task. These controls must be planned, but even the best planned testing program will languish if the inputs (requirements, design specifications, trace links, risk management artifacts) are missing or inadequate. Consequently, the validation plan must include assurances that those inputs will be adequate for the testing tasks. These assurances occur early in the life cycle of the software and are in the form of the review of and participation in the development of those inputs. It is a difficult, unpleasant, thankless task to try to make up for inadequate test inputs late in the life cycle. Worse, the results of testing based on sketchy inputs seldom, if ever, are as rewarding as when the validation activities have begun early in the life cycle to assure that the test inputs are in place when test designs begin.

Much of the content of the prior chapters has been devoted to documentation. The documentation takes the form of specifications filled with requirements, design detail comprised of design level requirements, and trace links to help us understand how requirements are fulfilled by designs. Of course, for maintenance of the device, it is good to have some written documentation of how the device works, and what the perceived requirements were that resulted in the design. However, prior to release of the device to the clinical environment, the driving force behind this documentation is to prepare the design outputs to make the testing effort as efficient and effective as possible. At least that is true from the perspective of this book on software validation. Developers might also conclude that the purpose of the documentation is to prepare the design outputs to make the implementation effort as efficient and effective as possible. That, too, is true, and the objectives are not independent of each other. If the intermediate design documentation outputs are effective in making the development of the software more efficient and effective, they also will make the testing of that software more efficient and effective. The inverse of this is also true.

So what exactly about the documentation is important for the testing task?

An essential element of a software test case is the expected result. It is the key detail that permits objective evaluation of the actual test result. This necessary testing information is obtained from the corresponding, predefined definition or specification.... The real effort of effective software testing lies in the definition of what is to be tested rather than in the performance of the test.

—*General Principles of Software Validation, Section 5.2.5*

Exactly. It is the need for expected results in testing that is at the root of this. One cannot know what to expect of a device's behavior unless those expectations are rooted in documentation. In any effective test, the expected behavior is called out in the test procedure. Where did the test procedure author get the information so that he or she knows what to expect? Hopefully from written documentation, or at least from spoken folklore specifications. The *last thing* one wants is for the test creator to define "expected behavior" as whatever passes the test. In other words, if tests are written to pass observed behavior as a substitute for expected behavior, the end result is that the tester makes the ultimate decision of what behavior is acceptable, not the team of developers, marketers, clinical specialists, regulatory professionals, or any other stakeholder. The bulk of the effort; that is, the thinking, which should go into testing, should be in the development of the test cases and procedures, and not in ad hoc, random, undocumented testing. Test development is made more efficient and effective if the design outputs used as inputs to the test development are of high quality.

## Why We Test Software

Most reasonable people would agree with the FDA and that testing of software is an imperfect activity. In other words, one can never, through testing, prove that any piece of software is free of error. One can prove that there *are* errors by discovering them. However, if errors are not found, that does not prove that there are no errors. It simply proves that the testers did not find any errors.

So, why are we bothering with testing if we cannot prove the device is free of error? Validation in general and testing in particular increase our confidence in the software. After all, if we find any defects or errors in our testing activity, the software benefits and our confidence should increase, assuming the defects or errors are corrected.

That makes sense. More testing should increase our confidence more than less testing would. Likewise, the more errors and defects that are found, the more confident we are in the software, right? Not so fast—it's not that simple. Consider two devices of equivalent complexity. The testing of Device A resulted in the identification of just two defects. Device B's testing effort resulted in 2,000 recorded errors and defects. Which device would you be more confident in? Well, it depends. It depends on how good the testing was, and how all those errors were managed to completion and verification of the corrections. Any testing program for a device of moderate level of complexity with only a few reported defects is suspect. Testing projects for moderate complexity devices routinely identify at least hundreds, and more often thousands of defects in the course of the project.

Isn't it odd that one might have more confidence in software for which thousands of errors were found than in software which had only two identified errors? Maybe the developers for Device A were really that much better—a thousand times better—and just did not make mistakes. I have encountered a number of truly excellent software developers, but none that could write software that was defect-free after being independently tested. Granted, many defects and errors have to do with requirements issues in which the software is perfectly implemented, but the result is not acceptable for the intended purpose of the software and device. So does that mean that the software testing project that uncovers many errors and defects means we should be confident in the resulting software? To know for sure, one would have to look at the details. How significant were the defects and errors that were recorded? How effectively were the errors corrected, and how many other errors were injected and subsequently discovered in the course of correcting earlier generations of errors? Were all of the defects that were serious enough to correct actually corrected, and were those corrections independently verified? Did the rate at which new errors were being found ultimately taper off to near zero at the end of the project?

Evidently, there is more to building confidence in software than logging a lot of errors, or not logging errors, or spending a lot of time testing the software. The extra ingredient is our confidence in the processes and procedures that were used to generate the test results and to resolve issues identified through testing. Confidence in the design and development processes further boost our confidence that the testers had complete and accurate information upon which to base the tests. That is exactly why so much of this book is devoted to validation activities that are focused on activities other than testing. The quality of the test effort will always be improved by good design and validation processes and procedures.

## Defining Software Testing

The following is a definition that I have used for some time. It is long, but it includes the key tasks and nuances that agree with current thinking about software testing for the regulated medical device industry:

Software testing: An activity in which a system, subsystem, or individual unit of software is executed under *specified conditions*. The *expected results* are anticipated from requirements or designs. The *actual results* are observed and recorded. An *assessment* is made as to whether the actual results are acceptably equivalent to the anticipated results.

The four italicized parts of this definition lead us naturally to an understanding of what a software test is. A software test is comprised of:

1. *Specified conditions*. These are the procedural steps that a tester will follow to set up a test for the specific values, conditions, or other inputs that will test a specific feature or requirement of the software.
2. *Expected results*. The test inputs set up by the specified conditions are selected to test a specific behavior of the software. The correct expected

behavior of the software is known by the test author from the software documentation (i.e., requirements and designs).

3. *Actual results:* The observed results may be a simple pass/fail check box that the expected result was observed, or may be numeric values, graphics, or images produced by the device. Regardless of what form the actual results take, they must be in a form that needs little interpretation in the assessment that follows to determine whether the test passed or not. This is how the FDA's need for "objective evidence" is fulfilled.
4. *Assessment:* A well-designed test makes assessment as simple as possible. The tester should be able to assess whether a test passes without the need for an expert to tell him or her whether it did or not. Of course, things don't always go according to plan, and sometimes someone with more experience with the software or with testing processes may need to be consulted. For example, consider a test step in a test procedure, a shown in Table 13.1.

In most cases this is very clear, and the sensor reads within 0.1 degree of 0 Centigrade (Pass) or not (Fail). The "evidence" is "objective" and the assessment is trivial. Suppose, however, the tester observes that the sensor reading is flashing, or that the reading was 0 degrees, but would flicker intermittently to 0.2 degrees. These are situations in which the test has not anticipated these behaviors, and the assessment may need a more experienced person to interpret the results. In the first example, if the sensor is supposed to flash at freezing temperatures, the test should pass, but the test should probably be updated to mention the flashing to avoid future confusion. In the second example, the test reviewer might fail the test; praise the tester for his/her observational skills, and consider updating the test to indicate how long the sensor holds the temperature in the valid range to pass the test.

Some examples of types of testing that do not strictly adhere to these definitions include ad hoc testing, session-based testing, and some types of user testing. These types of testing by their very nature do not have detailed specified conditions, or sometimes even a record of actual results. These kinds of "tests" are, nonetheless, valuable in finding software defects that more formal tests overlook. More will be said about these later in this chapter.

### Testing Versus Exercising

Testing software is quite different from testing mechanical or electrical hardware. In an earlier chapter the differences between software and hardware were examined. At that time it was pointed out that software defects in medical devices are almost

**Table 13.1** Sample Test Step in a Test Procedure

<i>Specified Condition</i>	<i>Expected Result</i>	<i>Actual Result</i>
Place temperature sensor in an ice and water solution. After 2 minutes read the Centigrade temperature from the sensor.	The sensor should read 0 degrees Centigrade $+/- 0.1$ degree.	<input type="checkbox"/> Pass <input type="checkbox"/> Fail Suspect observations: <hr/> <hr/>

always related to design failures. In other words, software does not wear out the more we use it the way a mechanical solution might. Consequently, when testing software the tester needs to think about every pathway through the software, the range of potential inputs, every decision point, and every calculation. Each of these needs to be tested by challenging the software, not exercising it.

A hardware mechanism, such as the brakes on an automobile or a mechanical brace for a human knee might be tested by putting it on a test fixture that operates the mechanisms or joints thousands or even millions of times with appropriate instrumentation attached to monitor the performance of the mechanism under test. These test fixtures are often called exercisers. They are designed to operate the device under test under normal operations repetitively to discover failure modes that might develop as a mechanism wears over time. With few exceptions, this approach is not very productive in testing software for all of the reasons mentioned above. Although this approach is sometimes used for testing software it is usually not productive unless used for specific purposes such as finding errors related to timing or other random events.

So, why even mention exercising software? It is mentioned here because when designs only cover the normal cases, they are simply “exercising” the software. The chances of finding a defect are fairly remote when testing only the normal, anticipated operation of the software. Most defects are found in the “dark corners” of the software. That is defects are most commonly found with uses that may not have been anticipated, sequences of events that are outside the normal use cases, inputs that are outside the usual normal range of inputs, and at the boundaries between normal and abnormal usage.

Good testers often anticipate what the developer may not have anticipated, and develop tests that intentionally try to break the software to uncover defective behavior. One might ask what value there is in testing software for inputs that are far outside the range of expected inputs, or may not even be of the expected type (for example testing for the behavior of the software when an alphanumeric entry is made in the field is clearly numeric). The simple reason is that when the software is put in the hands of hundreds or thousands of users in the field, unexpected things happen. Keys may be pressed accidentally. It could also be that what may seem like obvious usage to the software developer is not at all obvious to a new user. The only thing worse than unexpected behavior from a user is unexpected behavior from the software in response to unexpected behavior from the user!

## The Psychology of Testing

Long ago, I read a book on the psychology of computer programming. The book, [1] (still in print), had sections on programming as a group activity, and programming as an individual activity. It examined the ego, personality, intelligence, and motivation of the programmer to provide a glimpse into what made them tick. Of relevance to this text is thinking about development and *validation* as a group activity.

The relationship between developers and testers, or the validation team in general can be a contentious one. The organization must arrange for the testers and developers to be independent of each other, yet they need to recognize that they are

still teammates working for the same common goal. That is often easier said than done, especially when the two teams are very independent and/or geographically separated.

The independence is needed because the tester should not be influenced by the developers about what they should and should not test. The testers should not report and be responsible to the development organization where they could be put in the position of being responsible for delaying the development schedule. Testers, by virtue of their success in finding defects, may be blamed for holding up the software release simply because they keep finding problems. The wrong thing to do under such circumstances is to pressure the testers to pick up the pace (i.e., find fewer defects) or worse, to punish the testers for finding more defects.

The root of the problem is that the test team is successful when they find software defects that are often (though not always) inserted by the developers. The more successful the testers are in finding defects, the less successful the developers look. Consequently this often leads to a work atmosphere in which the developers are not eager to share information or assist the test team in their search for software defects. Likewise, the testers are reluctant to share their test plans with the developers simply because the testers want to take credit for finding defects. This is the conflict between individual and group success.

The reluctance to cooperate is just one symptom of an organization that is suffering from “validation-itis.” Developers whose work has been the subject of many defect reports may get demoralized, and may quit trying to produce error-free software. Instead, they may respond by making half-hearted attempts at developing the software only to hand it over to the testers prematurely. When the developers do not participate in pretesting the software, the end result is seldom as reliable as when they do. Depending too much on the testers is to put too much responsibility on the testing activity which itself is not perfect.

The role of test management is to develop processes, train the test and development teams, and continually reinforce to all that validation and testing are successful when they find defects. That is, finding defects is a good thing. Each defect found during development and testing is one more defect that will not make it to the field. Test processes should be designed so that both testers and developers are viewed as successful when a defect is found, traced to its root cause and effectively eliminated.

Developing a good attitude during software testing is not easy, and it is not a onetime event. It takes continuous reinforcement and creativity to keep both teams working together cooperatively.

One thing that contributes to bad attitudes during testing is the perceived unidirectional criticism of the software by the test team. To balance this, one might consider using the developers as reviewers of the test designs. This accomplishes at least three things:

1. It balances the flow of criticism so that developers critique test designs and testers critique the software. When managed properly (i.e., not allowing it to turn into a revenge contest), this can keep the proceedings more humane.
2. The review of test designs by developers improves the test coverage considerably. Knowing that their software will be critiqued by the testers,

- the developers tend to take advantage of their opportunity to critique the tests, which generally works to improve the test coverage or design.
3. The review of test designs often results in improved understanding of the software design, and what it took to for the developers to get it to work. It is not unusual to hear comments like, “This test is OK, but it is really light on the part we had the most trouble developing. We had a lot of trouble with the XYZ algorithm. You might want to cover that a lot more.” Testers who take advantage of that kind of information are focusing their effort on areas of software that are more likely to fail. This is one component of the risk-based validation and testing that was covered earlier in this text.

## Levels of Testing

The GPSV describes three distinct levels of testing:

In order to provide a thorough and rigorous examination of a software product, development testing is typically organized into levels. As an example, a software product’s testing can be organized into unit, integration, and system levels of testing.

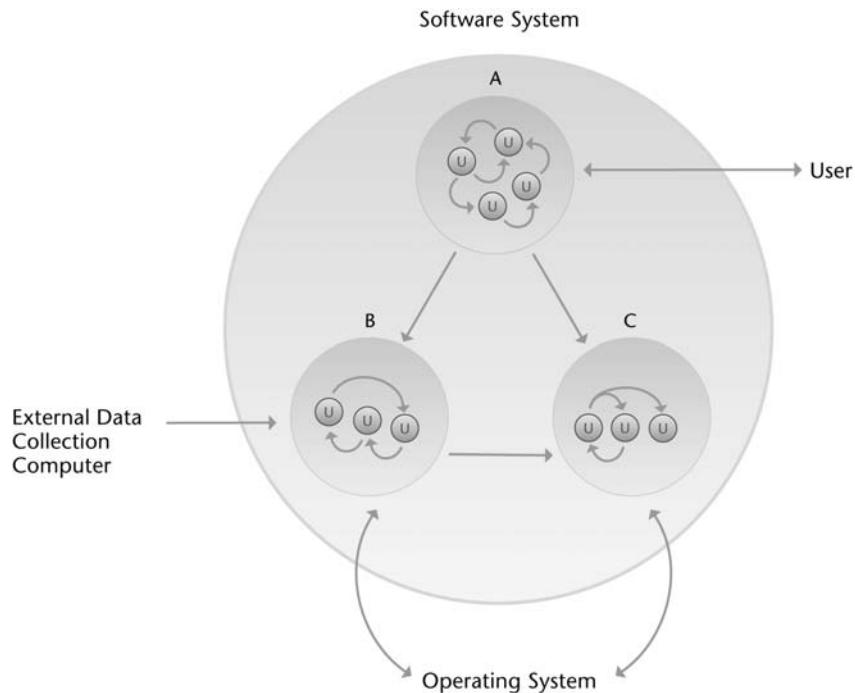
- 1) Unit (module or component) level testing focuses on the early examination of sub-program functionality and ensures that functionality not visible at the system level is examined by testing. Unit testing ensures that quality software units are furnished for integration into the finished software product.
- 2) Integration level testing focuses on the transfer of data and control across a program’s internal and external interfaces. External interfaces are those with other software (including operating system software), system hardware, and the users and can be described as communications links.
- 3) System level testing demonstrates that all specified functionality exists and that the software product is trustworthy. This testing verifies the as-built program’s functionality and performance with respect to the requirements for the software product as exhibited on the specified operating platform(s). ”

—*General Principles of Software Validation: Section 5.2.5*

Figure 13.1 shows the relationships of the three levels of testing. System-level testing is testing the entirety of the software as represented by the large outer circle in the diagram. System-level testing (also commonly referred to as black box testing) is usually done with no knowledge of the details of the design of the underlying software, based only on the system-level software requirements.

Figure 13.1 shows that the example software system is comprised of three subsystems, A, B, and C. Each of the subsystems is further comprised of units, as represented by the small circles with “U” inside.

Each of the units has interfaces with other units within the subsystems as represented by arrowed lines. The subsystems similarly interface with each other and with external (and not necessarily software) subsystems. Each interface is represented by an arrowed line. Integration-level tests are tests of the subsystems through the inter-



**Figure 13.1** Representation of three levels of software testing.

faces. For example, Subsystem A in the figure could be tested by supplying inputs from the user, and monitoring the outputs that are communicated to Subsystems B and C in the diagram. One could think of this kind of testing as black box testing of a subsystem without knowledge of how the units are designed to meet the subsystem requirements. In fact, knowing that subsystems behave predictably would be valuable before integrating them—hence the terminology “integration-level” testing. Very complicated interfaces (like user interfaces or computer communication protocols) are deserving of their own sets of requirements and tests of those requirements and might also be considered as integration level tests.

Unit-level testing is also referred to as white box testing, module testing, or structural testing. This level of testing takes advantage of detailed knowledge of how the software is designed and implemented at its most atomic level. The point of unit level testing is to test each “unit” separately from the others in the software system. The diagram indicates that the software “units” also interface with each other as represented by the arrowed lines connecting units. Testing those interfaces might also be considered integration-level testing. However, except for very complex interfaces, testing the interfaces between individual units is usually covered by the unit-level testing.

### Unit-Level Testing

This frequently involves the use of software unit test tools, or the creation of a separate collection of custom software that tests device software unit under test (UUT). These tools work by:

1. Passing test inputs into the UUT;
2. Checking inputs passed from the UUT to subordinate units, and simulating responses from the subordinate units in a predictable way that tests the UUT;
3. Collecting the actual outputs from the UUT;
4. Comparing the actual outputs and behavior with behavior predicted from the design documentation;
5. Documenting what was tested, the anticipated results, and the actual results.

The guidance stops short of defining exactly what a unit of software is for unit-level testing. That makes it important for test planning to define a software unit in the context of the project. Typically, a unit is equivalent to a software function (i.e. a function in the context of a computer language), or object, although it is not unheard of for units to be defined at a lower resolution (i.e., a collection of functions or objects that are logically related).

Regardless of the definition of the unit, the objectives of unit-level testing are the same. Unit tests should be more thorough in testing the structure of the software than can be done by system-level testing with its limited knowledge of the design and its limited control over exercising specific pathways in the software.

Let's take a break from the details of unit testing for a moment to remind ourselves what we are attempting to accomplish with unit-level testing, and why unit-level testing is a recommended approach. Earlier chapters spent a lot of page-space covering the definitions of, and relationships among validation, verification, and testing. Unit-level testing is a verification test activity, the objective of which is to verify that the software units: once implemented, meet the design intent as defined in the design documentation: that is, the design requirements. The design requirements should have been based on the software requirements, which, in turn, should have been traceable back to system requirements, which ultimately were ultimately rooted in user and other stakeholder needs.

Why bring that up now? There are two important reasons to be reminded of the roots of software testing:

1. Without proper requirements and designs that are traceable back to needs, testing becomes merely exercising software. If a tester doesn't know what the functional intent is from software requirements, or doesn't understand the structural intent of the software as implemented, the tests that are developed and executed are little more than a game of chance that the tester will come across something that he or she thinks is suspect. The comprehensiveness of coverage and value of the test results is highly correlated with the quality of all the design outputs that preceded the testing. That is why so much of this book on validation has focused on the artifacts that precede testing in the life cycle. It goes without saying, but if those design documents are not in place before the tests are created, they are of limited value for testing. Design documentation will still have value as a maintenance tool, but its value as a validation tool is almost worthless if it is written as retrospectively after the software and tests are complete.
2. The second reason to remind ourselves that unit testing is a verification activity is that it is far too easy to get immersed in the details of what makes

for a good unit test, and forget the reason that we are unit testing. It is quite possible to write a unit test that has great coverage metrics, and is above average in every attribute that describes a unit test, but is still a poor test. As some of those attributes are discussed below, remember that they are simply tools that can be used to create or measure the quality of a test. They are not the end result or goal of unit testing, they are the means to the goal which is to verify accurate and correct implementation of the software design, which is traceable and has been reviewed for conformance with the requirements.

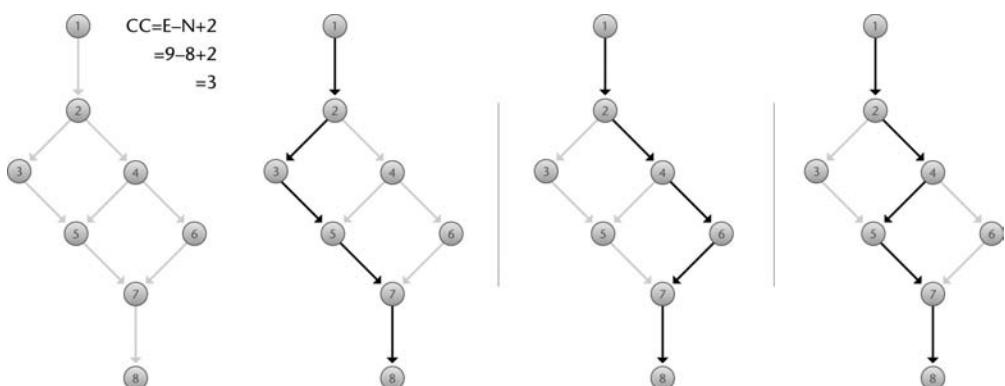
### Unit-Level Testing and Path Coverage

Some of the measurable attributes of unit tests include branch, statement (or instruction), and path coverage. Certain unit-level test tools can automatically calculate coverage metrics based on the tests designed. These metrics are generally metrics for how much of the source code is *exercised*. They are not indicative of how well the tests are designed. These metrics are a measure of the breadth of coverage, not the depth or quality of coverage.

One can think of a software unit as a collection of nodes and edges as shown in Figure 13.2. A node (represented by a circle in the figure) is a branching point that is usually representative of some conditional instruction in the software. The edges (represented by line segments in the figure) are the possible software pathways that result from each branch emanating from a node in the software. Branch coverage, then, is the ratio of the number of branches (i.e., edges) exercised in the software unit to the total number of branches in the unit. Statement coverage is number of source statements (or instructions) exercised to the total number of executable statements in the software unit.

### McCabe Cyclomatic Complexity Metric and Path Coverage

Path coverage is slightly more complicated than branch coverage. A path (or pathway) through the software unit is a unique path from the entrance point of the software to an exit point, selecting one branch at each node. The total number of



**Figure 13.2** Calculating McCabe's cyclomatic complexity.

paths through a given software unit is easily determined by a software metric called McCabe's cyclomatic complexity software metric. Cyclomatic complexity (CC) is often used as a metric for determining the complexity, or quality of software (assuming highly complex units are of lower quality). CC can also be used as a metric of path coverage for unit-level tests. There are a number of automated software complexity measurement tools commercially available that will calculate the CC metric, but the basic calculation is:

$$CC = E - N + 2$$

where

CC is the cyclomatic complexity;

E is the number of edges;

N is the number of nodes.

Referring again to the example of Figure 13.2, the simple software unit represented by the leftmost diagram has nine edges, eight nodes, and thus a cyclomatic complexity metric of three ( $9 - 8 + 2$ ). The three possible unique paths through the software unit are shown highlighted in the rightmost three diagrams of Figure 13.2. Although this figure represents a software unit with several if-then-else conditionals, one can readily appreciate that the CC calculations work equally well for multibranch conditionals (such as switch/case statements in the C language), and for loop structures such as while-loop or for-loop structures.

It should be readily evident that a high path coverage metric for unit testing would be a significantly greater booster in confidence in the software than simple branch and statement coverage. Path coverage will result in branches and statements being exercised several times for the varying path conditions.

Unfortunately, a high path coverage metric does not automatically translate to high confidence either. The path coverage metric only indicates what percentage of the software paths were *exercised*. It still doesn't tell one how well the paths were *tested*.

Are we making too much of this difference between exercising and testing software? Consider a nonsoftware example of testing a new tire just coming out of the research and development lab. We could mount the tire on a car, inflate it and if it doesn't go flat it passes the test. Would you be ready to put it on your car? We could start the car drive forward a distance equal to one tire circumference. There—complete “path” coverage—every inch of the tire has been exposed to the pavement. Now are you ready to buy one? You probably would not have the confidence in the tire until you knew the new tire had been tested through temperature extremes, pressure extremes, impact extremes, load extremes, velocity extremes, and whatever else keeps tire experts awake at night. The point is that there is a big difference between exercising and testing, even when the exercising is broadly comprehensive and measurable.

One might naturally think that the closer these unit-level test metrics are to 100%, the better. That is true. One's confidence in software is enhanced with quantifiable metrics indicating that the unit-level tests have at least caused a high percentage of the branches, pathways, and statements to be executed. At least one would be

more confident than if large percentages of the software had not been executed. However, these metrics can be misleading and can instill false confidence unless one more closely inspects the tests for their depth of coverage.

To drive this point one step farther, consider a simple software unit that selects and executes a simple calculation based on a simple conditional test:

```

if a>b
  then c = sqrt(a)/b
  else c = sqrt(b)/a
endif

```

Figure 13.3 shows this software unit in a graphic form. The unit has a CC of 2 (check this on your own from the formula above). In other words, there are only two pathways through the software: one down the left branch and one down the right branch. One could exercise the software with two pairs of values for “a” and “b” that send the software down both branches. That gives us 100% path coverage, and by the way, 100% branch and statement coverage, too. On the surface the metrics would give us some confidence since all the software has been *exercised*.

However, the software skeptic might try a few other sets of values such as those shown in the table of Figure 13.3, turning the exercise into more of a test. Of the 10 pairs of numbers tested (i.e., test cases) in the table, only 5 of the 10 pass! Had testing only used the first two test cases, one might have concluded (incorrectly) that the software was fully tested since there was 100% unit-level test path coverage, and no failures had been seen. This kind of flawed analysis could lead one to a false sense of confidence.

Developing tests is much like developing the software itself. Careful consideration and review of the test designs and the selection of test cases coupled with good coverage statistics builds well-deserved confidence in the software. This applies to unit-level testing, and all other levels and types of testing as well.

Before leaving the topic of unit-level testing and complexity metrics, let’s examine a few other ways to leverage McCabe’s CC metrics, and also look at some of the

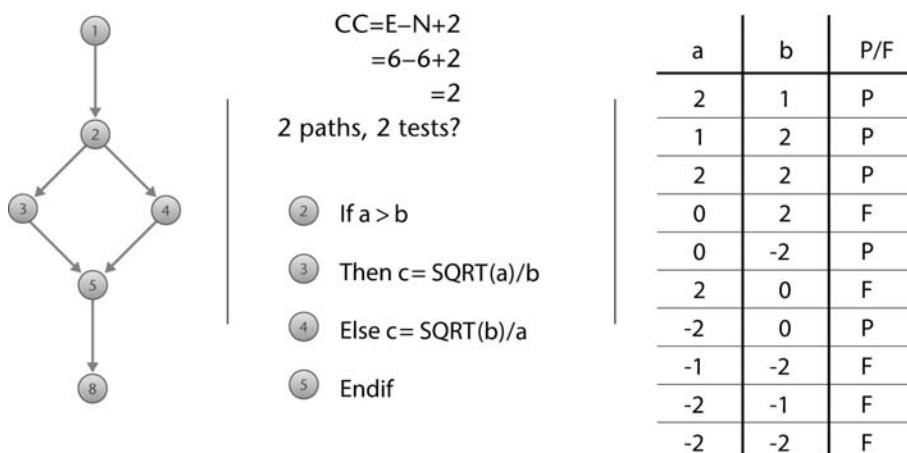


Figure 13.3 Path coverage is not equivalent to thorough testing.

limitations of CC. First, let us be clear that McCabe's CC is not the only software complexity metric that is widely used. It is just one that is widely known, intuitive, and easy to use for managing software development and testing. Other metrics won't be discussed in this text, but the literature is plentiful, and automated metrics tools are available for only one or two hundred dollars—well worth looking into!

From the “square root” example above, it is evident that unit-level testing can involve a lot of tests and test cases to build a high level of confidence in the software. This simple if-then-else example could have benefited from quite a few more test cases than were shown. Imagine how complex the unit-level test would be if a software unit had a CC of 50! In practice, it is common to find software units with high complexity metrics (CC and others) before unit-level testing is started. High complexity translates to a high likelihood of defective behavior, and a lot of expensive testing to find it all. Consequently, it is recommended that the developers take another look at the highly complex units to determine if they could be redesigned to reduce complexity, or could be subdivided into smaller units each of which would be of more manageable complexity. The end result is that the software is more likely to be reliable once the complexity is reduced, and the testing that will support that conclusion is simpler, less costly, and more likely itself to be comprehensive.

That is how metrics can be used to deal with highly complex units. What can software metrics do help us with unit-level testing of the very simple units? Recent experience with third-party unit-level test projects has shown that about 40% of the software units in a typical medical device with embedded software are of CC equal to 1. Twenty-five percent of the software units have a CC of 1 and also have three or fewer lines of code! Many of those are simple assignment statements, or instructions to read a value from a sensor or digital register to return to the calling function.

In the real world in which there is never enough time or money for all the testing that could be done, use of metrics affords an easy way to focus the precious resources where they are most likely to produce results (another facet of risk based validation). It is very unlikely that a defect will be found in a unit comprised of single instruction assignment. Probabilities increase as computational complexities are added to the unit regardless of its CC. The probability of defective behavior also increases with the size of the unit in number of lines of code. Unit-level test methodologies that, as part of the unit level test plan, define criteria for what will be tested at the unit level, can reduce the overall cost of unit-level testing by eliminating the tests for those units that are highly unlikely to produce defects and that are perceived by testers as being a waste of time because of the overhead involved in documenting even single line units.

If that approach is to be used, then additional metrics should be used to supplement McCabe's CC. McCabe's CC is not an effective metric when used on languages that are less procedural than C. CC is also insensitive to the number of lines of code (size), and to nesting depth of conditional or loops. McCabe cannot tell if code is poorly structured; it only calculates its simple CC for source code no matter how well or poorly structured it is. Although a firm believer in using software metrics to plan unit level testing, I would never rely solely on automatically calculated software metrics as a substitute for at least a cursory visual inspection of the software to confirm what the metrics imply as a reason for reduced testing. After all, those metrics themselves are calculated by software.

### Other Software Complexity Metrics and Unit Test Prioritization

There are other metrics that are better suited for some of the shortcomings of the CC metric. A collection of metrics gives a better rounded impression of the software's likelihood of being defective. Since automated tools exist to calculate these metrics, there is little reason not to use several of them. A few that may be of interest for this use include:

- *The extended cyclomatic complexity measure.* This metric in addition to calculating complexity based on the number of execution paths also includes a factor for the complexity of the conditional test that determines which execution path is selected.
- *The Halstead length measure.* This metric computes complexity based on the number of operators and operands within a software module. That is, it provides a measure of the complexity of any calculations in the unit.
- *Lines of code (LOC).* This is a simple count of the number of executable lines of code, not including blank lines and comment lines. Number of comment lines per unit is also an indication of how well the unit is documented; that is, of how well it is understood.

Let's conclude the discussion of unit-level testing with some practical perspective on what unit-level testing can provide for building confidence in software. It should be clear that unit-level testing is a verification activity, and that design requirements are needed before tests are designed for it truly to be a verification activity. There should be little doubt or debate that testing to verify design requirements should be more productive in finding defects than in a less focused method of poking around with a debugger, especially if that "poking" is only done by the developer of the software himself or herself.

Design requirements cannot anticipate every possible defect of the software to be written, so testers cannot rely solely on what is written as requirement for their testing. They must also rely on experience and suspicion in pushing the limits of testing to find defective behavior. In the square root example above, the design requirements may not have anticipated zero values for the denominator, or negative values for the square root. Does that mean that the test designer shouldn't try those values? Of course not. For example, the test designer can reasonably expect that the software unit was not supposed to trap out on a divide by zero error. So despite the fact that the design requirement didn't specify what the software should do, the tester is within his or her bounds to test it even if the anticipated response is unknown. This will raise an issue with defect management in which the behavior should be included in the requirements, or the parameters for the calculation should be tested by the software itself for invalid values prior to the calculation. Of course, if invalid values are found in by bounds-testing the inputs, some expected response should be found in the design requirements for that result.

### Integration-Level Testing

We all kind of know what integration level testing is, but as mentioned early in this text whenever someone says they "kind of know" that really means "I don't exactly

know.” Part of the lack of clarity about integration-level testing stems from the fact that there is little guidance on what is expected for integration-level testing in the GPSV guidance. The GPSV does mention that “There may be a distinct integration level of testing.” That is far from being a strong recommendation.

The GPSV mentions integration-level testing in the context of test planning, policies and procedures, and test designs and test cases. However, the closest thing the FDA has to offer in the GPSV to define integration level testing is:

Integration level testing focuses on the transfer of data and control across a program’s internal and external interfaces. External interfaces are those with other software (including operating system software), system hardware, and the users and can be described as communications links.”

—General Principles of Software Validation, Section 5.2.5

The internal/external interfaces and the communications links to which the GPSV refers are represented by the arrows in Figure 13.1. Integration testing is the testing of those arrows. One can appreciate that for larger systems, these interfaces can become quite complex, consisting of multilayered hierarchies of software units and software subsystems. Coming up with an integration test plan is not a trivial task. The integration test effort should build confidence in the software by finding defects or providing assurance that defects could not be identified under rigorous test conditions. Not all of the interfaces or links are equally likely to build confidence. One might prioritize the integration test activities in the following priority order to focus the test effort on the most critical interfaces:

1. *Safety critical functions*: The interfaces for the safety critical software functions are probably the highest priority interfaces to test. That should include interfaces from sensor to software unit, between software units, and to and from safety subsystems.
2. *External communication protocols*: These may exist to external data collection devices, hospital networks, laboratory data systems, or electronic medical systems. It is difficult to know or control how information communicated externally might be used, so one must assume that the use of the data may be safety critical and these links should be thoroughly tested.
3. *Subsystem to subsystem interfaces*: Especially in multiprocessor systems these interfaces, normal and abnormal communications, normal and abnormal timing specifications, normal and abnormal ranges of data values communicated, all should be tested. Note that these communications might be digital (Ethernet, RS-232, I<sup>2</sup>C, etc.), but they might also be analog signals or data values communicated via shared memory or a database.
4. *Interfaces with the user(s)*: Users include patient and operator of the device if they are different. Much of this testing might be covered under verification testing of user interface requirements at the system level.
5. *Interfaces between individual software units* (functions, objects, etc.) should be considered for testing if the units are responsible for high-risk functionality. High-risk functionality is that which is harmful if it fails. That likelihood, as mentioned in Chapter 8 is related to many things, including complexity, size, skill of developer and so forth.

Integration testing is almost totally unique for each project undertaken. That is one reason that the integration test plan is so important. It is difficult to ad hoc test at the integration level with any degree of comprehensiveness. That also makes it difficult to provide examples of what integration tests look like, or even how they are designed—there just isn't any that are typical.

### Device Communications Testing

Perhaps as close as we can come to a typical integration-level test is the testing of medical device communications protocols. Those protocols differ greatly, but testing them does have some commonality. Communications testing is like other forms of verification and validation testing in that a successful test needs to anticipate the correct behavior. In other words there must be a collection of communications protocol requirements<sup>1</sup>. As with most verification efforts, detailed, testable requirements are perhaps the most important ingredient for success. These specifications (i.e., collections of requirements) are also called interface requirements specifications, communication protocol specifications, or interprocessor communications specifications.

These specifications should include high-level descriptions of:

- Communication timing requirements;
- Message initiation and acknowledgement sequencing;
- Details on message structure (envelope, synchronization patterns, error checking and correction, addressing, sequence numbers, etc.);
- Handling of unexpected or out-of-sequence messages;
- Command groups;
- Error detection, reporting, and recovery;
- Resynchronization details.

Additionally, if applicable, a command dictionary should be part of the protocol specification and should include:

- The structure of each command;
- Data elements within each command (i.e., data types, valid ranges), and handling for out-of-range data elements;
- System events that trigger each command (e.g., alarm messages versus routine heartbeat messages triggered every  $n^{\text{th}}$  second);
- Acknowledgment details (ACKs) and recovery from negative acknowledgments (NAKs);
- Command-specific requirements for timing.

Finally, a good communications specification also includes use of case-based data-flow diagrams that show typical communications sequences that are anticipated.

1. The description of medical device communications testing is summarized from Vogel, D. A. and D. S. Bernazzani, "Communications Protocol Testing: Balancing Technology, Planning and Compliance," Medical Electronics Manufacturing, Autumn 2004.

pated in a normally functioning system, as well as in adverse system conditions such as dropped communications, nonsense message receipt, alarm conditions, system power up and down, and restart.

Once adequate requirements are available, the integration level communications testing turns to planning how to test each of those categories of requirements listed above. Communications (Ethernet, RS-232, or RS-422, etc.) with external devices or computers can be monitored and sometimes tested with commercial off-the-shelf test equipment. More often than not, these devices are communications monitors, and, while useful for examining the communications traffic on the communications medium, they are limited in their ability to insert errors to test the handling on both sides of the communications. In other words they are good monitors, sometimes good exercisers, but seldom good testers.

Development systems, specifically debuggers and in-circuit emulators, are often used to test communications software. The procedure is to set breakpoints in the communications software to stop the system so that the test engineer can view the contents of a buffer that is full of information received or about to be transmitted. This approach is useful for some testing; however, it is labor-intensive and time-consuming, and it requires a skilled tester. This approach also is limited in its ability to test a system in full operation at its stressed timing limits.

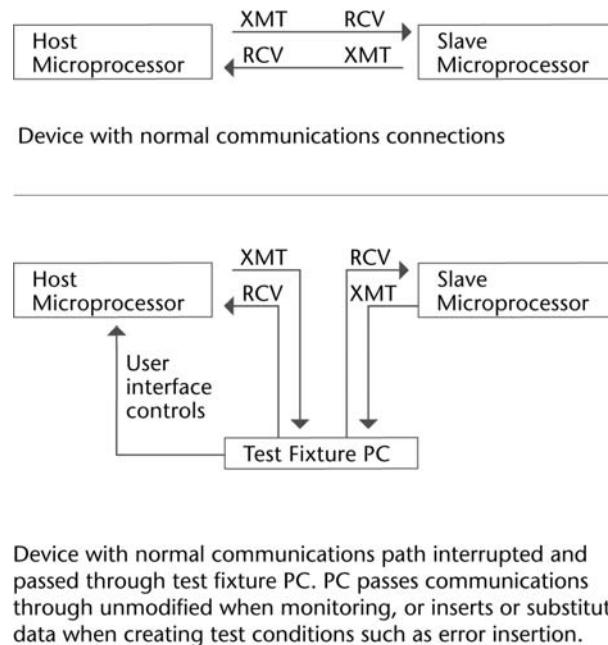
Most communications protocols require rapid sequencing of messages and timely responses that make testing difficult when manually testing through debugger breakpoints. Often the testing is confined not only by the technical limitations of the method, but also by the difficulty of setting up test scenarios, thus making it difficult and expensive in manpower to capture and examine sufficient data.

A more general approach to communications testing that works well for external communications—as well as intradevice interprocessor communications—is to insert a test computer (usually a PC) between the two processors of a given communication. Figure 13.1 shows this configuration for two processors that communicate bidirectionally on separate transmit (XMT) and receive (RCV) lines. The test computer intercepts transmissions from each microprocessor and relays them to the intended receiving processors.

This configuration gives the test computer the opportunity to check the validity of the communication against the expected result, or to insert errors to test the error-recovery requirements. Generally, the test computer can intercept and relay communications data quickly enough to keep the protocol timing within specifications. Additionally, timing delays can be inserted into the communications exchanges to test the software at its specified timing bounds. The test computer in the communications path can easily simulate dropped messages, out-of-sequence messages, unexpected messages, and so forth.

The arrow in Figure 13.4 labeled “user interface controls” indicates that the test computer functionality is often extended to enable it to control—and even to monitor—the user interface of the device under test. This control makes it possible to create scripted tests that run and collect results automatically as directed by the test computer.

Why is this important for communications testing? It certainly adds some complexity to the task to develop such a test system. Major advantages of this approach include:



**Figure 13.4** Block diagram of a communications protocol test setup.

- *Testing that can't be done without instrumentation:* For example, some tests may require complex sequences to set up for the actual test situation. Using a debugger to set breakpoints and force values for the setups would violate timing specifications, thereby making it impossible to test the interface software untouched. The test computer can be scripted to automate the setups in a real-time environment with fully featured communications software that honors timing specifications.
- *Timing analyses:* Some timing analyses can be done using an off-the-shelf communications monitor; however, a test computer can be scripted not only to set up the scenarios for testing key timing specifications, but also to log the maximum and minimum delays encountered during a test run. The test computer also can insert out-of-specification timing delays into a system to test the system's response to the delay and its ability to recover from such a delay.
- *Regression testing:* Testing communications by manual means is time-consuming. There is a temptation to cut back on communications tests on regression runs simply because they take so much time. Small changes in the device software can significantly affect system timing, which can, in turn, greatly affect the communications timing. An instrumented communications test system can be automated to a large extent to allow automated or semiautomated regression runs of communications tests as the device software evolves.
- *Long-run tests:* In some systems, communications events happen infrequently, requiring long tests to capture enough events to complete a meaningful test. Automated scripts accomplish this nicely with minimal manual labor. Testing professionals recognize that the one parameter that most closely correlates with finding defects is the number of hours of test time on the device

under test. Automating the test process to cycle through a number of use scenarios repeatedly over a long time period can be valuable. Often, it enables the testing engineer to find defects that would otherwise show up only in the field.

- *Stress tests:* Stressing communications is also difficult—if not impossible—to do by manual methods, but it is perhaps the most likely scenario in which the communications will fail. An instrumented test system can be used to stress a communications link because it pushes communications at a maximum rate for extended periods. It also inserts error conditions that stress the error detection and recovery processes that further use up bandwidth to deliver the same content. Test systems implemented with user-interface controls can further stress the system by driving the user interface at speeds that would be difficult for a user to sustain for any reasonable period. This stress can help flush out timing problems between user-interface stimulated communications and the communications timing.

One can appreciate how unique integration testing is for each device's architecture from that description of device communications testing. With that, let us move on from integration-level testing to testing at the system level.

### **System-Level Software Testing**

System-level software testing itself can be divided into several subcategories based on the device (i.e., system) characteristics are specifically being tested. However, what they all have in common is that the tests are black box tests. That is, they were derived from expectations of the software's behavior found in software requirements specifications, system requirements specifications, or even user documentation, but not on design-level documentation or an examination of the code itself. To do so could result in a system-level test that verifies that the software does what it does (uninteresting result) rather than verifying that the software does what it should do (the whole point of testing).

The FDA's guidance on system-level testing in the GPSV recommends:

System level software testing addresses functional concerns and the following elements of a device's software that are related to the intended use(s):

- Performance issues (e.g., response times, reliability measurements);
- Responses to stress conditions, e.g., behavior under maximum load, continuous use;
- Operation of internal and external security features;
- Effectiveness of recovery procedures, including disaster recovery;
- Usability;
- Compatibility with other software products;
- Behavior in each of the defined hardware configurations; and
- Accuracy of documentation.

—*General Principles of Software Validation, Section 5.2.5*

Those “functional concerns” almost get lost in the itemization of other elements of system-level testing, but they are likely the most voluminous, and obvious types of system-level tests. Testing of functional concerns or just functional testing is the major verification test activity to take place with the software. Functional testing verifies correct implementation of the software requirements. It is implicit is that the functional behavior be described in adequate enough detail in software requirements that the functional test can anticipate the results of a test case. This is needed so that the tester can objectively determine if a given test passes or not.

At this point it is appropriate to attend to terminology so that the reader is not overwhelmed with too many terms and concepts that blend into a milieu of jargon. We describe testing in terms of levels, (unit, integration, and system) that relate to the level in the software architecture that is being tested. We can think of this as *where* we are testing. The above quotation from the GPSV describes “elements of the software related to intended use.” One might also describe those elements as attributes, qualities, characteristics, or even features of the software system to be tested. This is *what* we are testing. Shortly, different test methods and means for testing (vs. exercising) software will be covered. This is *how* we are testing.

Why bring this up now? We are about to look at a number of different terms in a condensed form. It is important to understand how these terms relate to each other to construct a meaningful test plan for any device software.

Why? Think of building an airplane. If one were using a new type of structural beam, or wing material, or engine turbine material, wouldn’t you expect that it would have been tested significantly before putting it into the airplane? Testing the material itself in the lab is the equivalent of unit-level testing. Testing the wing or engine alone in a controlled environment is equivalent to subsystem- or integration-level testing. Testing the interface (i.e., the attachment and controls) between the wing and airplane or between the engine and airplane is interface testing or in GPSV terminology is part of integration testing. Finally, taking the entire plane out for a test flight is the equivalent of system-level testing. Hopefully, it should be clear to anyone why it is not a good idea to start with system-level testing, or limit testing to only system-level testing for an airplane. So, why would that be appropriate for a medical device? It’s not.

The table in Figure 13.5 breaks the terminology down into the wheres, whats, and hows of software testing. Although much more could be added to the table, it clarifies the relationships of the terms to be used in this text, and will serve as a foundation for the reader to create his or her own table of test terms that are appropriate to the specifics of his or her own software being tested for its intended use in the medical device.

It should be obvious that not all methods are applicable to all software items, nor do all of the elements apply to every application. In fact, it is even possible that not all levels of testing may be appropriate for all software items.

The above quotation from the Section 5.2.5 of the GPSV suggests that the bulleted elements are addressed by system level testing, and they should be. However, in all likelihood, the GPSV author probably did not mean to imply that those elements are *only* covered by system-level testing. Take as an example recovery procedure testing. Although some of those procedures may well be at the system level, they also may be at the integration/interface level, and may be able to be tested

Levels Of Testing Where It Is Tested	Elements (Attributes, Qualities, Characteristics) That Are Tested What Is Tested	Methods Of Testing How It Is Tested
Unit Level	Performance	Boundary Value Testing
Integration Level -Subsystem Level	Stress Security	Equivalence Class Testing Sampling
-Interface & Communications Level	Recovery Procedures Usability	Calculation Accuracy Testing
System Level	Compatibility Configuration Documentation Alarms, Alerts, Service and Maintenance Codes Accuracy Reliability	Error Guessing Ad Hoc Testing User Case Testing Session Based Testing Error Insertion

**Figure 13.5** Testing terminology.

only at the unit level where the fault condition can be forced. The point is that a good test plan considers what software system characteristics (i.e., elements), or parts of those characteristics will be tested at each level of testing. Within that level of testing for a specific characteristic, the test designer considers what methods best apply.

### System-Level Verification Testing Versus Validation Testing

The functional testing referred to in the GPSV quotation above is a verification test of requirements that have been explicitly developed in a software requirements document or collection of documents. It is verification that the individual requirements have, in fact, been correctly implemented in the software.

Validation testing is somewhat different from verification testing, although procedurally it may share test designs with other verification tests. Recall that device validation is to build confidence that the device satisfies the users' needs, while device verification provides objective evidence that the requirements and specifications have been satisfied in the implementation.

Validation testing of device software would normally be a system-level software test simply because it would be difficult to demonstrate that the needs of the user have been fulfilled without putting a finished product into the intended use environment with intended users.

Now tie in the above concept of "elements" of the device software that are tested in with verification and validation testing. Consider for example, the alarms and alerts elements of testing that are mentioned in Figure 13.4. Device alarms and alerts, as mentioned above, may be tested at one or more *levels* of testing. Since these alarms and alerts are likely to be very specific in their description in the software requirements, testing for the existence and expected behavior of the alarms and

alerts most likely would be considered a verification activity. That is based on the assumption that the requirements for the alarms and alerts are embedded within some software requirements specification at the time the test is designed. But now, consider one of the *elements*, such as usability, and consider what a test for usability might look like. Specifically, how would one know if one passed the usability test or not?

Since the results of such tests are likely to become subjective rather than objective, one might well imagine that the tests themselves would look very different. For example, a usability test might put a user through a number of different use scenarios. The results of that test might depend on the success of the user in completing the use scenario successfully, or in a prescribed amount of time, or even on a survey-like response that elicits an opinion about usability from the user at the end of the test sequence. Each of these, though more subjective than the usual functional verification test, still could have verifiable requirements embedded in the specifications. For example, “The procedure for loading a new cassette shall be simple enough to allow 95% of the intended users to successfully load the cassette in a clinical setting in less than 15 seconds with no formal training.” This looks like more of a system-level (SyRS) requirement, but it is verifiable with a statistical sampling of intended users. Nonetheless, this type of requirement and the corresponding test for it definitely have a different look from your usual run-of-the-mill functional requirement and verification test.

The cassette loading example above is starting to inch toward a validation test; that is, one which tests whether the needs of the user, in this case the clinician, are met by the system design that may be partially implemented in software. Now, consider this example of a system requirement: “The system shall detect Lyme disease from the blood sample with less than a 2% false negative rate.” One can imagine that the system-level requirement might break down into any number of electronic, mechanical, chemical, and software requirements to achieve the specified accuracy. Each of those requirements may be verifiable under controlled laboratory conditions. However, to truly validate that the system requirement some methodical testing of the device in a clinical environment, with a variety of real patients, and a variety of intended users of the device will be required. This does not lead to the step-by-step verification test procedures that will be described later; rather, these validation tests most likely will be very light on procedural instructions and will depend upon observation of the intended users as they use the device, and analysis of the results against known “right answers” that may be confirmed by other means.

## Testing Methods

So far, we have discussed the *wheres* of testing (i.e., what level), and the *whats* of testing (i.e., the elements or characteristics of the software). In this section, we will examine some of the *hows* of testing by looking at some of the well-established test methodologies. There is not room in this text to cover each of these methodologies thoroughly, but there are many books available on software test methodologies.

Some of the old standbys that I have referred to over the years are listed in the bibliography at the end of this chapter.

One of the precepts of any type of testing is that one should test not only the normal and expected inputs or conditions but also the abnormal or unexpected inputs and conditions. Why? Why would one want to test alphabetic characters as inputs to a numeric data entry field? Why in the square root example used earlier in this chapter would one want to test input parameters that lead to zeroes in the denominator or square roots of negative numbers? The reason is that funny things happen in the real world. Users do not always do what they should do or what you expect them to do. Even other software developers do not always check that they are passing viable inputs off to a subfunction. One just cannot count on everything working the way it is expected to work. Software should be designed defensively so that it deals with the unexpected by expecting it, or at least detecting the unexpected and dealing with it in a safe and sensible way. Software tests should be designed to challenge the software with unexpected inputs to be sure that they are dealt with appropriately.

Some of the following methods are simply different ways of organizing the normal and abnormal test cases. Others are more analytical. This list is far from comprehensive, but many of the books referenced at the end of this chapter devote significant portions of their books to test methodologies and are excellent references for brainstorming new ways to test software.

### **Equivalence Class Testing**

Sometimes a software function is blessed with a finite number of combinations of inputs. An example of this might be software function that locks or unlocks an access door to a dangerous part of the device based on inputs from three binary sensors. In this case, since there are only three inputs and they are all binary, we know that eight test cases will completely cover all possible inputs combination.

Unfortunately, not all of software testing is that tidy. Suppose that a different software function requires five inputs for a calculation, and those five inputs are decimal numbers with 12 digits of accuracy extending over 10 orders of magnitude. In this case, it is unlikely that we would be able to test the function with every combination of all values of the five inputs. In situations like this, we can only test a limited subset of the possible values and combinations. The trick to successful software testing is to choose the *right* subsets that are most likely to find errors.

Other test methods to be described below will describe means for identifying test cases that are more likely to result in the identification of software errors than others. The method of equivalence class partitioning is to analyze the universe of possible test inputs for groupings and subgroupings that would be likely yield the same test result. For example, if testing a square root calculation, one might think of positive numbers and negative numbers as two equivalence classes. If one or several positive numbers return accurate square root answers, then one might conclude there is a high likelihood that all positive number inputs would return appropriate responses. Likewise, if one negative number is rejected with an appropriate error condition, then one might conclude that all might be rejected. This, of course, is not a guarantee that all inputs positive and negative would work, but it does suggest that

one might use some intelligence and critical thinking about the software being tested to divide the inputs or expected responses into groups to reduce the amount of testing that would be required by random input selection.

Just because the universe of inputs is divided into two classes doesn't mean the job is done. Equivalence classes are not mutually exclusive. The classes can overlap, or be subsets within other classes. In our square root example, one might also divide the possible inputs into perfect squares and nonperfect squares or inputs greater than one and less than one. The inputs 0 and 1 are yet another class of inputs whose square root is equal to the input itself. There are probably other groups that can be identified, but one can appreciate two major advantages to this approach:

1. The mere act of thinking critically about what the equivalence classes might be makes one think critically about where the weaknesses might be. Those weaknesses might be related to stressing the software, or underlying libraries, or the weaknesses might simply be suspected problem inputs that may not have been anticipated by the developer.
2. The existence of the equivalence classes allows one to focus testing in those classes and on the boundaries of those classes where the defects are most likely to be found. The net effect is that the likelihood of finding errors is higher, and the amount of testing needed is lower than if random sampling were used.

Equivalence class partitioning at a higher level can also be effective in testing complex user interfaces. In the following example, equivalence class partitioning is used to organize testing of GUI pop-up constructs used to communicate with the user of the medical device about alarms, alerts, and other device notifications. The above two advantages of equivalence class testing are leveraged: reduction of the testing overhead, and focusing the test effort where it is most likely to be productive. This example also leads us to the introduction of a new concept: "gray box" testing. Gray box testing is partly white box (unit-level, structural) testing and partly black box (system-level) testing. In other words, gray box testing can be thought of as system-level verification testing of requirements based on some knowledge and assumptions about the underlying structure of the software itself.

Consider the example of a fairly complex medical device with a graphic user interface (GUI) consisting of roughly 250 different screens on the GUI display, and about 300 different alarms, alerts, and device notifications of varying priority. As one approaches the test plan for the alarm/alerts/notification testing one realizes that there is a very large number of potential combinations of GUI screens and special pop-up constructs to handle the alarms, alerts, and notifications.

One approach to testing is to break these special pop-up constructs into the three equivalence classes of alarms, alerts, and notifications. Each of these classes has its own unique set of behaviors. If one *assumes* that the operational behavior of all alarms is handled by the same software function or functions, then perhaps the detailed testing of the operational behavior of alarms could be handled in one set (protocol) of test designs. Likewise, two other protocols might be designated for detailed testing of the operational behavior of the alerts and notifications. The key word in *italics* here is *assumes*. The validity of the rationale for testing the

operational behavior on a limited number of screens (perhaps even just one screen) is based on the assumption that the handling of that operational behavior is handled by the very same software instance for the pop-up class. Since that assumption is so critical, and especially so for things like alarm handling which is critical to the safety of the device, that assumption itself needs to be tested. That is what makes this gray box testing; someone needs to verify that the code is structured in such a way that the assumption is met. If the assumption is not met, then the test planning needs to be redesigned.

So, how does one keep track of these assumptions? They actually need to be written as requirements either in the software requirements themselves, or in the design requirements with some traceability linkage to the test designs so the requirement is not altered or eliminated in later versions of the software. Some remnants or artifact of the requirement to meet that assumption is necessary for this type of equivalence class partitioning to continue to be valid in subsequent releases of the software.

There also are a number of requirements related to the prioritization of handling each of the three types of events in the example when they exist concurrently. This leads to seven other equivalence classes:

1. Alarm alone;
2. Alert alone;
3. Notification alone;
4. Alarm and Alert concurrently;
5. Alarm and Notification concurrently;
6. Alert and Notification concurrently;
7. Alarm, Alert, and Notification concurrently.

Now, since the prioritization handling might be affected by the order in which the events occur, these concurrent classes could be further divided as:

1. Alarm followed by alarm;
2. Alert followed by alert;
3. Notification followed by notification;
4. Alarm followed by Alert;
5. Alert followed by Alarm;
6. Alarm followed by Notification;
7. Notification followed by Alarm;
8. Alert followed by Notification;
9. Notification followed by Alert;
10. Alarm followed by Alert followed by Notification;
11. Alarm followed by Notification followed by Alert;
12. Alert followed by Alarm followed by Notification;
13. Alert followed by Notification followed by Alarm;
14. Notification followed by Alarm followed by Alert;
15. Notification followed by Alert followed by Alarm.

This process of analysis continues to break down the classes into separately identifiable categories. For instance, what happens for an alarm, alert, alarm sequence? What happens when two events occur simultaneously (if that can be defined)? On it goes; testing of alarms and alerts can be very detailed, and given how critical they are to safe operation, they should be. There are three points relevant to this example of equivalence class testing:

1. Testing is reduced by recognizing the similarities in behavior combinations and testing those combinations only once—not on each of the 250 screens with each of the 300 different alarms alerts and notifications.
2. The process of identifying the equivalence classes leads to the discovery of combinations that may not have been anticipated, implemented, or exercised by the developer, or were overlooked in the requirements or development process. This analysis leads to more complete broad coverage than if one randomly sampled test cases.
3. The assumptions that legitimize the equivalence classes are articulated, documented, and are themselves verified to make the equivalence class test reduction valid.

There is one final thing to mention related to the equivalence class analysis. Several times above it was mentioned that this analysis is likely to uncover situations not anticipated by the developers. That probably means that they were also not anticipated by the requirements from which the developer presumably worked. (They did use the requirements didn't they?) This also means that if the scenarios weren't anticipated in requirements or development, then the expected behavior may be unknown. Now what?

These are situations that need to be addressed at the requirements level. The appropriate behavior needs to be determined and/or approved by those who originally approved the requirements (not the tester alone, and not the developer alone). Although this seems cumbersome, and may lead one to think that is just delays the development process, exactly the opposite is true. The alternative is that the software is blindsided in test with test scenarios never considered. Nobody is sure if the results are defects or not—nobody ever thought about it (but the tester). Regardless, time is wasted debating the disposition of the findings. If changes to the software are required, design documentation also needs to be updated, as will the requirements. Tests may need to be altered. Changes to software may lead to major architectural changes. Testing will need to be repeated. Major changes late in the life cycle are one of those risk factors that increase the likelihood of errors. Regardless how distasteful it is to face issues brought up by this analysis early in the life cycle, it is guaranteed that it will be much worse if ignored until final testing.

### **Boundary Value Testing**

Without calling it by name, boundary value testing has been referred to a number of times in this chapter. When the testing of normal and abnormal cases, or testing at the limits of acceptable ranges has been mentioned, they were really types of boundary value testing. Boundary value testing and equivalence class partitioning are

closely related. The limits, or boundaries of identified equivalence classes are precisely those boundaries that should be carefully tested.

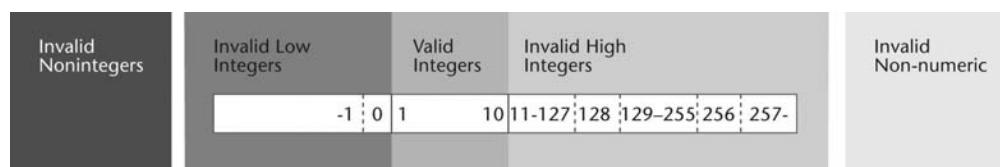
Consider a data input field that specifies that valid entries are “integer values from 1 through 10”. Figure 13.6 shows the relationship of the equivalence classes and boundaries for testing. This figure shows major equivalence classes for valid integers, invalid integers whose values are too high, and invalid integers whose values are too low. There are shared boundaries on either end of the valid range. Also shown are two equivalence classes that have no shared boundaries with the integer value classes. Those are the invalid numeric but nonintegers and the invalid nonnumeric inputs.

The figure also shows a segmented band meant to represent subclasses within the major equivalence classes. For instance, in the invalid low integer class, there is a dotted line boundary between 0 and –1. The conclusion one might draw is that because 0 and 1 are numbers with such different properties in calculations, perhaps both should be tested. On the high invalid integers there are boundaries between 10 (valid) and 11 (invalid), but also around the values 128 and 256. Why? This is an error guess that the input may be converted to a numeric stored in an 8-bit signed or unsigned variable. If so, then a test value of 129 (signed) or 257 (unsigned) could be treated the same as a 1, changing the response of the software from treating the inputs as invalid (correct behavior) to valid (incorrect).

Why are we concerned about boundaries? Boundary conditions are those which experience has shown are the productive test conditions for finding software errors. The reasons that this is true include:

- Functionality on the “abnormal” side of the boundary, or just close to a boundary may have been overlooked by developers and requirements authors.
- Equivalence classes, as we saw in the preceding section, can overlap or be subsets of each other. These complex boundary conditions and subequivalence class boundaries are even more complex and less likely to have been considered in requirements or design. Consequently, the likelihood of finding errors in testing is correspondingly higher.
- Testing the boundary conditions forces the tester into considering the normal and abnormal cases. Boundary analysis, (i.e, the identification of the boundary conditions) in itself promotes critical thinking needed to identify those test cases that are most likely to identify errors.

So, what is meant by boundaries? First, recognize that boundaries can be applied to inputs to the software under test or to the outputs or behaviors of the soft-



**Figure 13.6** Relationship of equivalence class partitions and boundaries.

ware being tested. All should be considered in equivalence class partitioning and in the testing of the boundaries around those equivalence classes. As with so many other activities that have been discussed in this text, boundary value testing does not subject itself well to a checklist approach. It is not difficult, but it does require thinking about the software under test in a rather unique way that improves with experience. A good, professional tester may build and maintain his or her own list of potential boundaries that should be considered as part of an evolving checklist. As a guideline to getting the rookie tester thinking like this, the following list of common boundary conditions that are encountered with software is a good starting point, but by no means should be considered a comprehensive list.

1. Any numerical range of valid inputs or valid outputs defines the boundary between acceptable and unacceptable. For example, if the range of acceptable inputs is specified at integer values from 5 to 10, good boundary test values might be 4, 5, and 6 on the low end and 9, 10, and 11 on the high end. The same idea applies to the outputs of the software. For example, the output may be specified as a “positive value greater than zero.” To test the bounds of the output, the tester would supply inputs to the function to attempt to drive the output to a number incrementally larger than 0, 0 itself, and incrementally smaller than zero.
2. In unit-level, structural testing, boundaries might apply to boundaries of physical memory as in buffers. If a buffer is 256 bytes in length, the tester might consider test conditions that supply inputs that are of length 0, and 1 at the low boundary and 255, 256, and 257 bytes long at the high end to test the appropriate behavior for inputs for the equivalence class of input lengths that fit the buffer, and the class of input lengths that do not fit the buffer.
3. Boundaries exist in performance levels. Consider a surgical device that rotates the cutting edge in three speed ranges by substituting gears. The performance of the device at the high and low speeds at each range would be excellent targets for testing to verify that the specified speed, torque, ambient noise level, and so forth are achieved in each range (equivalence class) at the high and low limits of each the range (boundary value testing).
4. Sometimes software behavior can be broken into discrete segments. For example if a report is to be generated with 45 line items on a page. Tests for proper pagination behavior may test 0 and 1 line of data, 44, 45, and 46 lines, and several multiples of {44, 45, 46} pages in length. Clearly one boundary is the full page length of 45. There is a boundary at the end of each page, hence the multiples of 45. One might also guess that the software calculates the number of pages of the report and uses that number as a loop counter to print all pages. Another test boundary could be at the boundary of 255 or 256 pages (times 45 lines per page = 11,475 or 11,520) Why? The tester might guess that the count could be stored in an 8-bit variable. When the number of pages exceeds 255, the count would wrap around back to 0, leading to unexpected behavior. Similarly, (although somewhat impractical) the tester may guess that the number of lines of data is store in a 15-bit signed or 16-bit unsigned number. That defines two more boundaries at 32,768 and 65,536 lines of data or 729 or 1,457 pages.

5. For sorted or ordered lists, there are boundaries at the minimum possible value and maximum possible value to test. However, there are also other boundaries that can be defined with a little analysis. Suppose a list of decimal integer and noninteger numbers is to be sorted in a column of a display or report. The display is 6 digits wide. Additional boundaries exist for testing such as:

Values 1, 2, ... 6 digits wide integer values;  
Values 1, 2, ... 6 digits wide noninteger values;  
Numbers >1 or <1 (possibility of not displaying the leading 0);  
Boundary between integer and noninteger values;  
0, 1, 2, ... 5 digits to the right of the decimal.

Of course there may be other identifiable boundaries, but even this limited list will result in a fair number of test cases and give fairly comprehensive coverage.

### Calculations and Accuracy Testing

The testing of complex calculations and algorithms like image analysis, feature recognition, or complex signal processing is deserving of a text devoted to that topic alone. There is a significant risk when medical devices embed significant efficacy or safety functionality inside complex calculations or algorithms. As the complexity of the software increases, the number of people who understand that part of the software decreases, and so does the depth of their understanding. The end result is that the test coverage reflects the understanding of the tester, and software that may be most critical to the device gets tested the least. Oddly, this is sometimes written about as “overtesting” of the better understood, simpler, and less critical software. (I guess everyone is just tired of hearing about undertest, so this at least gets peoples’ attention!)

We won’t even attempt to take on the technicalities of this type of testing other than to say that it often requires libraries of known signal or image inputs with known results or expected behaviors of the algorithm being tested. Detailed reviews are extremely valuable as are detailed unit and integration level testing of algorithms.

Unfortunately, a calculation doesn’t have to be too complex to be error-prone. Often that comes from just a lack of understanding of the software tools and libraries that are being used to do the calculation. Let’s take a look at floating point calculations. I have always found them to be fertile grounds for finding defects in devices.

A floating point number is usually seen in source code or printed results in the form:

xxxxx E yy

where the x’s are decimal digits known as the mantissa, and the y’s are the exponent of 10. The mantissa times 10 raised to the exponent power equals the number being represented. (At the binary level, these are all represented base 2, not base 10 which

can lead its own set of unexpected consequences.) The number of x's in the mantissa that a given version of floating point will support is the precision or number of significant digits. Sometimes the mantissa is displayed as “.xxxxx” or “x.xxxx”, but that does not affect the following analysis.

When multiplying or dividing, floating point numbers works just fine. Suppose a floating point library supports only six significant digits—that is six digits in the mantissa. Multiplying or dividing two numbers of six significant digit results in a six significant digit floating point number (i.e., rounded or truncated at the sixth digit). However, adding and subtracting floating point numbers can result in computational errors if one does not understand the limitations of floating point calculations.

**Problem 1.** Suppose a drug delivery device delivers two-thirds of a microliter of drug per step of the stepper motor that powers the pump. In a six-significant digit floating point library, that is .666667E-06 liters per step. The software calculates in floating point the delivered volume by adding the volume per step to the cumulative total volume for each step of the motor. When floating point numbers are added (or subtracted) they are, in effect, expanded into fixed point numbers to align the digits, added, then represented as a floating point result.

Figure 13.7 shows the effect of our limited floating point resolution on the calculation as described above. After only 15 motor steps, the pump actually infused only  $10\mu\text{L}$  of medication, but there is a 0.01%  $\{1 - (0.666/0.666667)\}$  error per step in the contribution to the calculation of total volume. The reason as shown in the middle column is that to align the significant digits for the addition, the lowest order two digits are beyond the resolution of the calculation. (The six significant digits are to the left of the vertical line. Those to the right are lost. To simplify the example, assume they are truncated, not rounded.)

Following the progression to higher total volumes, one can see that at 15,000 steps, or only 10 mL of delivery, the error per motor step is 10%. Even worse, after 150,000 steps there will be no contribution to total volume since all digits of the per step volume are to the right of the vertical line. When calculated this way, the total volume would never go beyond 100 mL!

Steps	Actual Infusion	Calculation Next Step	% Error Per Step
15	$10\mu\text{L}$	10.0000 00 E-6 .6666 67 E-6	.01%
150	$100\mu\text{L}$	100.0000 000 E-6 .666 667 E-6	.1%
1,500	1 mL	1000.00 0000 E-6 .66 6667 E-6	1%
15,000	10 mL	10000.0 00000 E-6 .6 66667 E-6	10%
150,000	100 mL	100000. 000000 E-6 .666667 E-6	100%

**Figure 13.7** Example of errors due to floating point addition.

**Problem 2.** Subtraction suffers from the same problems as addition if the two operands are orders of magnitude different in their value. There is, however, another problem with subtraction that relates to losing resolution (i.e., precision or significant digits) in floating point subtractions. This occurs when two numbers whose high-order digits are equal are subtracted from each other. For example:

$$654321 \text{ E}3 - .654198 \text{ E}3 = .000123\text{E}3 = .123\underline{000}$$

Because the high-order digits are equal, the high-order digits of the difference are zeroes. In floating point notation, the first nonzero digit becomes the most significant digit. The floating point difference shifts left and placeholder zeroes are shifted in on the right. The three underlined zeroes in the above are the three placeholder zeroes. Since they are only placeholders, there is no information in those three digits that tell us anything more about the actual value of the difference in what we get from the three high-order digits.

The result of that subtraction is fine if that is the final answer in the calculation. That is all the information we have to work with. However, if this difference represents an intermediate value, for example if this difference is to be multiplied by another number, the loss of resolution can result in misleading results.

Consider what happens if the above difference (.12300) is subsequently multiplied by another floating point value (.987654). The multiplication is done long-hand below in a rather unusual way to make the effect of those placeholder zeroes stand out. Lines a, b, and c show the results of the placeholder digits on the multiplication; they simply add zeroes. Any of the summed digits in line d that included in its sum one of the zeroes from lines a, b, or c is not a reliable value of the true product because those three zeroes were arbitrarily chosen as placeholders to complete the floating point notation. In fact, even the next higher digit is questionable since it would have included in its sum the carry digit in the preceding digits sum. Therefore, the underlined digits in line d are all “trash” values because their values are corrupted by the placeholder zeroes.

	987654E-6
	x 123000
(a)	000000
(b)	000000
(c)	000000
	2962962
	1975308
	987654
(d)	<u>121481442000</u> E-12

What we are left with in this example would be a floating point result of .121481E0, but really, we are only confident in the first three digits of this result (.121000E0). In any floating point arithmetic operation, the number of significant digits of the result is only as great as the number of significant digits in the operands of an arithmetic operation. This example shows why the nonsignificant digits of the result are not significant.

Any floating point number is only an approximation of the actual value of the number to the number of digits supported by the floating point library. Suppose in the initial subtraction above that the two operands are irrational numbers whose values cannot be represented in a finite number of digits (e.g., pi). If we extend those approximations and reveal another four digits, the result might look something like:

$$.6543212468 \text{ E}3 - .6541981234 \text{ E}3 = .0001231234 \text{ E}3 = .123123\cancel{4}000$$

This is the same operation on the same numbers as in the original problem 2. The only difference is that the floating point approximation to the actual values has been extended by another four digits. Since the operands still agree in the first three most significant digits, three digits of significance (underlined in the difference) are still lost, but now four more digits of the actual difference are available. In other words, the difference of these two numbers, accurate to six digits, is .123123. However, our result from using six digit floating point calculations can only give us a value of .123000.

There is plenty more to say about floating point calculations, but let us return to software validation before this turns into a book on numerical analysis or computing methods. Why go to this level of detail on floating point operations, or mention them at all? Recall in the discussion of risk management and software, the point was made that it is very difficult to quantify the probability of error in software. In that discussion, a number of risk factors were suggested whose presence, experimentally, have been found to be correlated with a high probability of software error. These risk factors included the experience and skill levels of the developers and testers, and whether the functionality of the software is well understood or not. All of these factors seem to intersect whenever floating point operations are used in medical devices. In fact, awareness of floating point issues was much greater in the days before calculators and spreadsheets reduced our awareness by routinely computing numbers to many more significant digits than the typical user would ever be interested in using or displaying. Testers should always be highly suspicious of software that computes in floating point for critical operations in medical devices.

Many devices need to include complex calculations. So what can be done to assure the accuracy of those calculations? First, floating point is not always a necessity for calculations; integer calculations can often be substituted that do not suffer from the same issues as floating point. If floating point is a necessity, one can extend the number of significant digits used by the floating point library to several digits more than are needed for calculation and display. In many cases the additional digits of resolution can push the inaccuracies down into lower order digits that are not of interest. However, one must be aware that some of the problems (like the subtraction problem) still manifest themselves even in calculations with more significant digits. For example, to floating point numbers of any resolution that agree in the first n higher-order digits will always lose n digits of significance in the difference. The question for the developer, reviewer, and tester is whether those lost three digits of significance are relevant to other calculations that depend on that result or to the application itself. The order of calculations can often be changed to avoid floating point problems (by subtracting last), or formulas can make use of identities to alter the computation in a way that avoids floating point problems.

In testing floating-point calculations, the problems are most often found in the boundary conditions; that is, in calculations that push the limits of extremely large values or extremely small values or operations on values that are very nearly equal or very different. Multiplication and division are normally safe operations in that a multiplication or division operation on two of n digits of significance will always return a result of n digits of significance. Propagation of errors or loss of resolution through chains of calculations also should be avoided and should be the subject of significant testing. Thorough testing of calculations, especially floating point calculations, is difficult from the system level alone. Examination and review of formulas and orders of operation often leads to the weaknesses in methods used that can be subsequently verified in targeted testing.

Consider for example the Problem 1 scenario discussed above. Discovery of those types of errors could come from analysis of the computational methods, analysis of the source code leading to suspected weakness in subsequent testing, or by simply testing for accuracy throughout various delivery volume scenarios. Note that the first two of these test approaches would not be possible without knowledge of the formulas and structure of the source code. The third approach is quite easily done by pure black box system-level testing based on suspicion that accuracy might vary according to the delivery ranges of the device. In this case, one would not know that the inaccuracies were due to computational error, but subsequent analysis in trying to repair the inaccuracy (at least hopefully) would lead to that conclusion.

### Error Guess Testing

Error guessing has already been mentioned a few times above. It is a very productive type of testing on the basis of errors found per test case, but that rate can vary dramatically with the experience level of the tester. The reason is that the list of error guesses that each tester relies upon is based on his or her own experiences of how often he or she has made a certain mistake, or how often they have run across the problem in testing.

Examples of error guessing that have already been mentioned include:

- Off-by-one (OBO);
- Buffer overrun;
- Floating point computation loss of resolution and accuracy;
- Use of types that are ill-fit to the needs of the variable (e.g., possible need to work with values >255, stored in an 8-bit variable).

In fact, the first two of the examples above are so common, security hackers use them as error guesses to test security and communications software to find vulnerabilities that can be leveraged to gain access to secure systems.

It is very difficult to coach one on how to guess what errors might exist. However, processes can be put in place to make others on the test team aware of what kinds of errors others are finding. Once an error is made, it is highly likely it will be repeated. Making the team aware of what types of errors others have found will facilitate using those past finds as error guesses for future testing.

Some errors may be specific to a single project. An example of this is when significant major changes are made late in a project. This tends to leave a lot of unanticipated loose ends that result in errors on a common theme, but specific only to that software and that late change. Other errors are more broadly found. There are many checklists of common programming errors for software developers. One needs only do an Internet search on “common programming errors” followed by the name of the language of interest to find these lists. A more generic listing is found in Annex A and Annex B of the AAMI TIR32 [2]. One might also imagine that an error guess list would contain errors that are common to a development team or product line.

Error guess testing is effective if done informally with each tester guessing based on his or her experience. Reviews of test designs and test cases allow others with more and/or different experiences to add to the errors guessed for specific tests. Even better and more effective would be a formal process for building and maintaining a list of error guesses made available to all test designers to access.

### Ad Hoc Testing

Ad hoc or *impromptu* testing often is not taken seriously, but experience has shown that it is quite effective for specific uses for which other testing is not so well suited.

What is ad hoc testing? Ad hoc testing usually refers to undocumented testing, meaning that it is not scripted or proceduralized. It is used for a specific purpose, or specific narrow functionality of the software. In other words, the tester is asked to “Test the XYZ calibration routine in the software,” not just “Go test the software.” It is driven mostly by the tester’s intuition and experience on where and how to find defects (i.e., it is mostly undocumented error guessing). Why would one want undocumented, unscripted tests? There are two main reasons: speed and randomness. Let us look these reasons separately.

*Speed.* Well, of course, everyone wants to finish testing as quickly as possible. However, we can’t sacrifice the thoroughness, repeatability, or consistency of documented, scripted, reviewed test procedures for the speed of ad hoc testing. Ad hoc testing is fast because it is not thorough or documented and thus is not repeatable or consistent. However, there is a place and time for ad hoc testing to supplement traditional documented test procedures.

The point of testing is to find defects, and the earlier they are found the better. Suppose, for example, some changes are made to the device software. That software is released for continued testing. Would it make sense to test that change at the bottom of stack of other procedures? Perhaps a test procedure has not been written yet to address the software change. It could be weeks before the procedure is written and execution is completed. In the meantime, that change may be defective and could negatively impact other test and development efforts. The change needs to be tested quickly to avoid potentially wasting time to find that the change was not implemented correctly, didn’t work as anticipated, or just caused too many other problems that weren’t anticipated. In these situations, ad hoc testing is performed by testers who are well experienced with the software and have a sense of where and how to find any latent errors. Despite the fact that the testing is not documented,

there is success in finding the problems early enough to avoid wasting any time on problems created by the change.

Do these tests remain undocumented? The functionality they cover may already be covered in documented tests. If, however, an ad hoc test detects an error that is not detectable from any documented test procedure, then a documented procedure should be created. Why? So that when the error is fixed, a procedure exists for testing the fix. Also, (this has been mentioned before, but it is worth repeating several more times) once an error is made it has a high likelihood of being made again. Future changes to the software in this area could result in the same error resurfacing. A documented test procedure should be in place to verify that it does not happen again (see “Captured Defect Testing” below).

*Randomness.* This seems like a strange word to be used in a positive way when describing testing. However, it too has its place. Formally documented (i.e., not ad hoc) test procedures are great for repeatability when a defect is found. A repeatable path is documented for the developer who sets out to fix the defect, and a procedure is in place for retesting to be sure the defect is repaired. That repeatability can also work *against* the test effort. It forces repeatability in the way a piece of software is tested by restricting the testers to the procedure’s script. However, many errors are the result of users who use the software in unexpected ways. They may misunderstand how the software is to be used, they may make input errors, or they may simply start exploring the software. As usage patterns become more random and less anticipated by the software and requirements developers, the more likely it is that unanticipated usage pattern will stumble upon defective software.

Some randomness is good simply because one cannot systematically cover every unanticipated random usage of the software. A diversity of test methods that balances systematic documented procedures and more random methods like ad hoc testing will always give a better result than focusing all testing in one methodology.

Since ad hoc testing depends to some degree on random events to identify defects, one cannot expect that defects from ad hoc testing will present themselves in the first hour of testing. Defects will be detected in random intervals, so the longer one explores and experiments with the software in an ad hoc way, the more defects one is likely to find. The same cannot be said for proceduralized testing. Unless the software changes, one would expect the same result from a formal documented test procedure whether it was run for 10 minutes or 10 days (barring memory leaks, uninitialized pointers, etc.). Experience has shown that it is operationally productive to keep testers busy running ad hoc tests in areas of questionable software integrity (i.e., we suspect problems because the code is new, has history of problems or some other clues) whenever there is some tester downtime.

### Captured Defect Testing

OK, I made up the name for this category of testing because I have not seen anyone else do it in a formal way. Don’t run to Wikipedia to look up more information on it! Occasionally, in the course of validation testing with intended users, or during ad hoc testing, or even during proceduralized verification testing, one stumbles across a defect that is not anticipated or for which there is no specific test to force it to occur.

Let's call these "captured defects"; they were generated randomly and "captured" in a defect report. Captured defects can also arise from the execution of a documented test procedure if the tester notices some unexpected behavior that was not the focus of the test. That behavior was perhaps not even documented in the expected behavior verification of the setup steps of the test procedure.

In the section on ad hoc testing, it was suggested that errors that are stumbled upon randomly should have test procedures developed subsequently that re-create the errors to assist the developer who will fix them and to assist the tester who will verify their repair. This applies to defects that are captured from any undocumented random source. In addition to being useful for the developer and tester, these test procedures, which we have dubbed here as captured defect tests, serve as a record that the defect once existed and how it was created. Why is that needed? Again: because a defect that occurs once is more likely to occur again sometime in the life cycle of the software than it was to occur the first time it was detected. Defects have high likelihoods of coming back to life as the software is modified and remodified.

These captured defect test procedures can be added to existing test protocols (i.e., collections of procedures) that target similar functionality, or they can be collected in a separate protocol of captured defect tests. We prefer the separate protocol approach. Each procedure in this protocol did produce a defect at some point in the life cycle, so there is a higher probability that this protocol will be productive in finding defects than a protocol that mixes one or two captured defect procedures with a dozen other verification procedures that passed every time so far. Furthermore, the captured defects produced by the procedures in the protocol are defects that escaped detection by the systematic procedures that attempted to trap them. This protocol becomes a prime candidate for regression testing as the life cycle progresses since it is such a concentrated collection of procedures that have found defects and are likely to again.

## Other Test Methods

There are many other test methods that can be employed that are too numerous to cover here. Session-based testing (SBT) is one method that has received some attention in the last 10 years. SBT shares a number of similarities with ad hoc testing but does have a more formal process to describe the method and does produce test result documentation. The test designs are the responsibility of the tester (i.e., the person running the test) so the results will vary based on the experience and skill of the tester. SBT claims to work well with software that is lacking in formal requirements. That may make it a good test method given less than optimal inputs, but does nothing to make up for the other validation activities that build confidence that user needs have been accurately translated into working software. It is all left in the hands of the tester.

Documentation testing finally has been recognized as an important part of the validation of the device software. User error is recognized as sizeable part of software related defects. Software user interfaces can be obscure and allow users to err, but if the documentation (user manuals, labels, instruction cards, etc.) is just wrong then the user is lured into erratic behavior.

The bibliography at the end of this chapter lists a small subset of the dozens or hundreds of books related to software testing that I have found useful, many of which do explore other methods.

## Test Designs, Test Cases, and Test Procedures

In Chapter 4, quality system processes and procedures were described for various aspects of software design, development, and validation. At this point, let us introduce some new terminology related to software verification testing, and discuss how that might be tied into a process or procedure for designing, developing, and executing software verification and validation tests. Those new terms are test designs, test cases, and test procedures.

The usage of these terms here conforms to the definitions in the IEEE Standard for Software Test Documentation [3]. These terms are used commonly in the industry, often with little distinction of definitions. There are, however, distinct differences and interrelationships, and understanding those nuances can lead one to a better understanding of test development processes.

Before taking on the terminology, let us first consider why it is advantageous to break a test into three distinct components. It has been pointed out a number of times in this text that the process of software validation very closely mirrors the process of software development. The same can be said of test development specifically. Since the bulk of the effort in developing a test is in detailing the procedural steps, it makes sense to document and review the test approach, specific inputs, and specific outputs before investing the time in detailing the procedural steps. In other words, breaking a test down into these three components can make the review simpler, more effective, and yet can reduce the overall effort that goes into test development.

To be clear, it is not necessary to segregate the designs, cases, and procedures into different documents although one could do that if desired. We have found it quite practical to include the test design and selected test cases in the preamble of the test procedure. In effect, our test procedures start out as designs and test cases, and once reviewed and approved, evolve into a full-blown procedure as the procedural steps are added at the bottom of the document.

Let us take a look at what goes into each of these components.

### Test Designs

The test design should be written in simple natural language (e.g., English) to explain to the reviewers and the future author of the procedure the test approach, rationale, and logic behind the test. The test design should include the following:

- A listing of or traces to the requirement(s) to be tested by the combination of this design and the selected test cases.
- Any assumptions about the state of the software under test at the beginning of the test. For example, does the test assume that the software has been reinitialized to clear out any historical states or historical data values that might alter the logic of the test? Another example is an assumption of where one is in the software when the test begins.

- A simple, short preamble to help the reviewer understand the context of the test.
- A simple, short description of what the test is to accomplish, explaining why it is an adequate test of the requirement(s) under test. Also list any inputs and expected results to be detailed in the test cases.
- A description of what it would take to pass the test without qualification. In most cases, this will amount to simply passing each step of the procedure. In less proceduralized tests, such as some of the validation tests described above, it may take a little more explanation of what constitutes a pass. This is needed to make the test results conform with the objective evidence regulatory recommendation that has been mentioned several times before in this text.

Hopefully, “short and simple” has been stressed enough for test designs. It is not the purpose of a test design to just translate a detailed procedure into prose. The purpose is to give a reviewer an understanding of how the requirement will be tested so that he or she can determine if the coverage is adequate. The design is also useful to the person executing the test. A short description of the test design that puts it in perspective helps testers who often get lost in many lines of procedural detail and lose sight of what they are supposed to be looking for in the test.

## Test Cases

One can think of the test cases as the paired test inputs and expected outputs or behaviors that result from those inputs. The test design for a simple data entry field might simply state that “The tester will bring the software to the xyz screen, and position the cursor on the abc data entry field that is the object of the test. The requirement is for the software to only accept numeric integer values from 1 to 10. Valid values will be tested for acceptance and invalid values will be tested for rejection and appropriate recovery.”

The test cases for this design might be a table as shown in Table 13.2. The short test approach explained in the design, and the specific values to be used in the test should be all that a reviewer would need to see to determine whether the test is adequate to fully test the requirement. Compare that to reviewing a lengthy procedure in which the test input values are embedded in procedural steps scattered over many pages.

There are second-order advantages to organizing tests around this test design/test case structure. Consider a device interlock that depends on three binary sensors. Since each of these sensors can be in one of two states, this immediately suggests that one would expect to see eight ( $2^3$ ) test cases, one for each of the possible combinations of sensor states. Thinking about the problem of designing a test for the interlock in this format of presenting test cases in a concentrated way not only assists the reviewer in checking that the expected number of test cases are present, it also assists the test designer in knowing how many test cases *should be* necessary.

Some test designs are so simple that there may be only one test case. However, if it is noticed that the majority of tests are materializing with only one test case, that may be an indicator that the tests are only testing the normal cases; that is, lightly *exercising* the software rather than *testing* it.

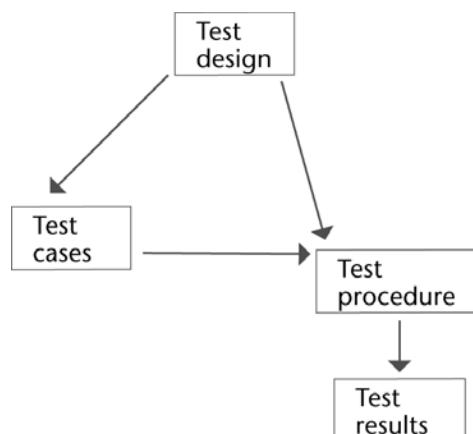
**Table 13.2** Example of Test Case Table.

<i>Test Inputs</i>	<i>Expected Result</i>	<i>Notes</i>
1	Result accepted	
5	Result accepted	
10	Result accepted	
0	Result rejected	
11	Result rejected	
001	Result accepted	
1.1	Result rejected	
.8	Result rejected	
-1	Result rejected	
Null	Result rejected	
a	Result rejected	
A	Result rejected	
%&	Result rejected	
111111 ... 1	Result rejected	If behavior allows, enter 257 1's to error guess buffer overrun up to 256 byte buffer

## Test Procedures

Figure 13.8 shows the relationship between test designs, test cases, and test procedures. The test setups, the assumptions to be met, the procedural steps needed to meet the test approach documented in the test design, and the actual test values used to test the software all come together in the test procedures. As the figure shows, the information from the test designs and the test cases are the inputs to the test procedure.

One might ask why test procedures are necessary at all since the presentation of test design and test cases are so easy to understand. Indeed, some types of tests that will be discussed below might in fact take that approach. However, for the run-of-the-mill functional verification test there are advantages associated with having detailed test procedures:

**Figure 13.8** Relationships of test designs, cases, and procedures.

- *Repeatability.* One of the most frustrating things for testers and developers is a situation in which the tester uncovers what is assumed to be a defect and the developer cannot re-create the defect so that he or she can fix it. Highly detailed test procedures have a better chance of leading the developer through the same sequence of steps that led the tester to the defect than if he or she is led to his or her own method of re-creating the sequence.
- *Scalability.* Frequently as projects approach their completion, there is a desire to aggressively push the testing schedule to complete the project as quickly as possible. Very detailed procedures allow relatively untrained testers to join the project late and contribute significantly without lengthy training. Step-by-step details of what the tester should do as well as details about what the tester should be observing and confirming allow the rookie testers to produce results of similar quality to those testers who have run the tests a number of times before.
- *High resolution testing.* Oftentimes the procedural setup for a software test may be comprised of a large number of procedural steps. Similarly, a number of test steps may be required for each of the test case inputs specified. If each of those steps has an observation that the tester is to confirm, it sets up a high-resolution process for detecting unexpected behavior in the software. Frequently the defects that are recorded in highly detailed procedures are not even related to the requirements that are the main object of the test procedure.

Of course, there also disadvantages associated with highly detailed test procedures. The level of detail is more costly to create than less detailed procedure methods. Similarly, the maintenance of highly detailed test procedures is more costly simply because the procedures become sensitive to any change in the software. For example, recall the example of the test for the data entry field. Imagine that five or six tester actions are required through the user interface to select the appropriate screen and position the cursor in the data field. Any change to the user interface that would affect those tester actions would require a maintenance action on the test procedure to bring it in line with the revised user interface behavior.

A complete example of the device requirements, test design, test cases, and test procedure is provided on the accompanying CD. Rather than burden the text with that level of detail, let us limit our discussion of test procedures to two high-level topics: the structure of the procedure steps themselves, and the cover sheet that travels with the test procedure as it is executed.

As mentioned above, we prefer to include the test design description and test case descriptions in the preamble of the test procedure itself. If a requirements management tool is used to create the tests and maintain the trace links to the requirements, one can use the tool to automate embedding the linked requirements themselves into the preamble.

The body of the test procedure is composed of the individual test steps. Early in this chapter, it was noticed that the test was comprised of four components: specified conditions, expected results, actual results, and an assessment of the results. Consequently, the test procedure might be formatted into three columns:

1. Tester action (specified conditions). These are the individual actions between observable events that are expected of the tester. They are short simple commands such as, “Power up the device” or “Click the Done button.”
2. Tester observation (expected results). These too should be short and simple, but in enough detail that the tester will not be confused about what he or she should be looking for. Note that these step-by-step observations may have little to do with the requirement that is actually the target of the test. The intermediate results leading up to the key result or results that verify the requirement serve at least two purposes. They are a confirmation that the test is proceeding as planned, and a verification that nothing unexpected happened in preparation for the key observations. In practice, a large percentage of the defects recorded in a test project actually come from these intermediate steps.
3. Assessment of results (actual results and result assessment). In the body of the procedure, a column is dedicated to the assessment of results. For a given procedure step, the cell might contain simple checkboxes for Pass or Fail, or a Yes or No to answer the question, “Are the actual results equal to the expected result?” As discussed earlier in this chapter, the actual results might require the tester to record a reading or data value that was observed as part of the test. It is always a good idea to record actual values; however, that alone is not “objective evidence” that the test passed. In addition to recording the data value, the tester should be required to assess whether that data value is within the expected range.

Assessing the results on a step-by-step basis is relatively straightforward. An interesting situation to consider is that in which one or more preliminary procedural setup steps fail, but the procedure steps for the key observation(s) for the requirement under test passes. Is that a failed test, or not? To answer more insightful questions like that, an overall test procedure result assessment is needed. We have found it practical to document that on a cover sheet that exists for each test procedure.

As with the test procedure example, an example of a test procedure cover sheet that we have used for some time is provided on the accompanying CD. What is relevant to this question is that the cover sheet requires a reviewer (usually of a higher experience level than the tester) to review all of the individual procedural test results and assessments and to draw a conclusion as to whether the overall test procedure passed or not. Those decisions may be influenced by personal preference, schedule pressure, and indecision. Unless that decision-making process is documented in a plan, it is unlikely that the decisions will be made consistently from reviewer to reviewer or from day to day.

Some of the things that might enter into the overall test procedure assessment decision include:

- Is the problem discovered by the failing step in the procedure; one that has been identified in another procedure? If so, the problem possibly is a duplicate, and there is little value in recording it as a new finding. There is also little value in failing the test overall if another test procedure (possibly one that was intended to test that functionality) has already failed for the same reason. If,

however, a particular procedure exercises the software in a way unlike any other test procedure (including the one that was intended to test the functionality) then perhaps the overall test should fail to be sure that it is rerun after the repair is made.

- Are test metrics being calculated on a per requirement basis or on a per test procedure basis? If testing progress metrics are being collected on a per procedure basis, failing an entire test procedure because one or more intermediate steps did not yield the expected results or behaviors will not lead to an accurate picture of how the testing effort is progressing. Imagine a misspelling in the user interface display. Hundreds of test procedures may need to exercise that display in the preliminary steps leading up to the key procedure steps for the requirements that they are testing. Therefore that one defect could result in hundreds of test procedure failures if the reviewer requires that every step of the procedure pass in order to pass the procedure overall. Likewise, once that misspelling is corrected, hundreds of procedures could pass. However, to come to that conclusion, those hundreds of test procedures would have to be reexecuted. Testing and retesting the repair of the same misspelling repeatedly is not effort that adds value or builds additional confidence in the software.

In general, a lot of test activity that yields little incremental benefit seems to result from failing an entire test procedure because one or more preliminary steps fail for reasons that are duplicated and tested elsewhere. On the other hand, it would look suspect if a high percentage of test procedures were marked as passes with failing individual steps. This may not meet the standard of objective evidence unless accompanied by some explanation of the assessment and resulting pass/fail decision. The place for that assessment is on the cover page.

## Managing Testing

So far in this chapter, we have talked about different kinds of tests, and different types of test methodologies. We discussed test designs, test cases, and test procedures. In this section, we will spend a little time talking about a few topics related to the management of the testing activities.

### The Importance of Randomness

Obviously, our goal is not to produce random test results. It is also not to come up with a formal, repeatable procedure with preselected random test inputs that are repeated every time the test procedure is run. What we are talking about here is a class of tests and methodologies that attempt to surface the unanticipated responses of the software to unanticipated inputs or usage patterns.

Earlier, on the topic of ad hoc testing, it was pointed out that the value of ad hoc testing or any other method using less proceduralized test designs were valuable because they injected an element of randomness into the testing process that was difficult to achieve through formal test procedures. That is true, and it is the role of the test manager who has the high-level view of the test plan in mind to be sure that

the random *and* formal elements of tests are balanced appropriately for the specifics of the device whose software is being tested.

Another way of injecting controlled randomness into the testing activity is by careful selection of who is developing the tests, and who is executing the tests.

Test developers tend to fall into patterns in their testing procedures. In fact, it is not unusual to see that copy-and-paste is used frequently in developing test procedures to cut down on the amount of manual labor involved in writing the procedures. The danger in this is that the procedures, although written perfectly, tend to exercise the software through the same pathways over and over again. So, although they are probably decent tests for the specific requirements that are being tested, by repeatedly putting the software through the same sequence of setup steps, any opportunity for randomly bumping into an unexpected error is eliminated.

One managerial way of dealing with this is to not assign an entire block of functionality to a single tester, but to have several testers working on the same section of functionality to introduce randomness. This is not to say that there is duplication of effort. If a section of the SRS is comprised of 60 requirements, one might assign 20 requirements to each of three test developers. In this way, even though each of the test developers may fall into his or her own pattern, at least there is a chance that they will be three separate patterns.

Similar issues arise during test execution. Testers experienced with the device may have run and rerun the test procedure a number of times during the course of the project. Boredom can set in, and the tester becomes less alert to unexpected behavior than someone new to the project. A tester who is new to the project or even new to testing at all, tends to question everything that is not documented, understood, or looks suspicious to them. At least, that is how they behave if properly prepared for the responsibility that goes with running the tests.

A tester may be the last one who has a chance to notice a problem that could injure or even kill the patient. Testers need to be made aware of this, and experience shows that, when made aware, they respond appropriately. Experienced testers are just as responsible and serious about their work when made aware of the implications of allowing a defect to go unnoticed. Their awareness can simply be worn down by repetition. They also may become accustomed to seeing defective behavior since early in the testing, and have mentally accepted it as normal, where a neophyte might not.

Experienced testers will have a deeper understanding of the requirements and the history of the project, probably giving them a better sense of error guessing than someone with less experience. The important thing is diversity. A test team full of young inexperienced rookies is not the goal, nor is a team full of testers who have worked on a project for three years. It is a balance of experience and inexperience that gives the best results.

## Independence

One final note for test management on the topic of who should be running the tests has to do with independence. It is very difficult if not impossible for one to be totally objective when testing one's own work. Consequently, some level of independence from the actual design and implementation is desirable.

The GPSV comments on this more broadly in the context of independence of review. However, it is clear that these comments apply to all verification and validation activities including testing:

Validation activities should be conducted using the basic quality assurance precept of “independence of review.” Self-validation is extremely difficult. When possible, an independent evaluation is always better, especially for higher risk applications. Some firms contract out for a third-party independent verification and validation, but this solution may not always be feasible. Another approach is to assign internal staff members that are not involved in a particular design or its implementation, but who have sufficient knowledge to evaluate the project and conduct the verification and validation activities. Smaller firms may need to be creative in how tasks are organized and assigned in order to maintain internal independence of review.

—*General Principles of Software Validation, Section 4.9*

It is not at all unusual for a software developer when asked about a suspected software error to respond with, “Well, that’s just how it works ... it’s not a bug.” *How it works* is not necessarily the same as *how it should work*. This is an example of how one’s perspective of one’s own work is quite different from the perspective of an independent reviewer or tester. Independent testing is always more objective.

A special case often comes up with unit-level testing. Unit-level testing is very technical, and usually (if not always) requires a software developer to design, develop, and execute the tests. Since software engineers are always in short supply, companies often allow developers to unit test their own software before integrating it or submitting for integration. 30 years or so of experience has never seen this work effectively. Either the testing just never gets done, or it covers the code inadequately. If nobody reviews the unit-level tests, this kind of testing slides by undetected. Even reviewing these tests probably requires a software developer, so the time saved in allowing a developer to test his or her own code is lost in having another developer review the tests.

It is also common (especially in agile life cycle models) for developers to pair up for the purpose of reviewing and testing each other’s code. This is better than self-testing, but familiarity with the partner and/or his or her work can lead to lax testing patterns that start to slip back to those approximating self-testing. With proper oversight, this method works quite well because the familiarity can also be leveraged for targeted error guessing.

## Informal Testing

We have found it useful to distinguish between formal testing and informal testing, which we sometimes refer to this as “dry run” testing. By informal we do not mean the same thing as ad hoc. The test results documentation is less formal because this kind of testing is done early in either the development or validation life cycle or both. In these early stages, apparent defects may be the result of software errors, but they could also be the result of test errors (in the designs, test case selections, or procedures). They might also be the result of requirements errors, or just preliminary test releases of software that have not implemented all of the requirements yet.

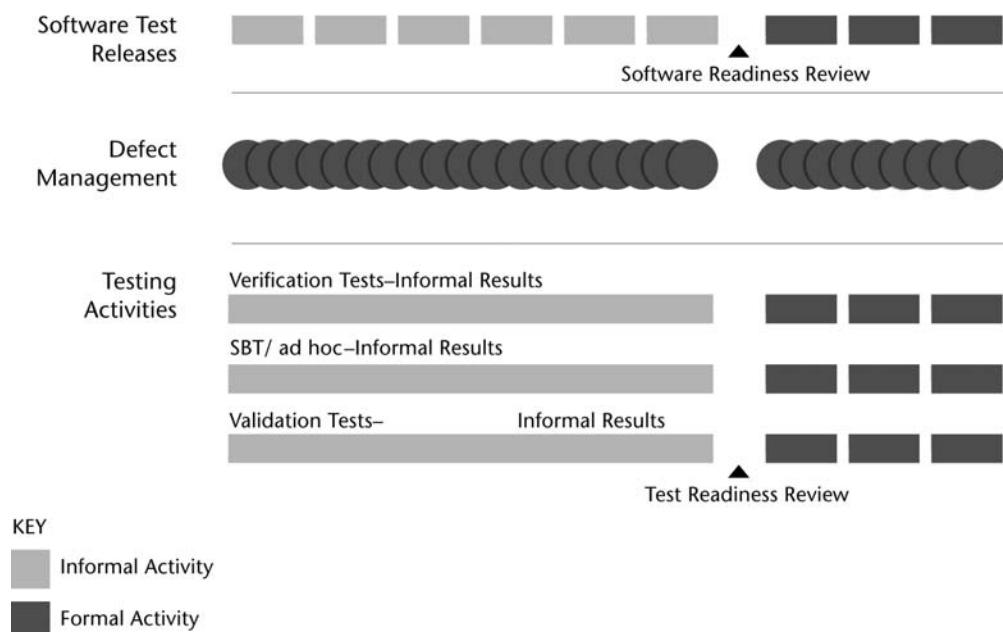
There is no value added to the quality of the product or the design history file to impose a formality to the documentation, collection, and organization of test results for software and tests that everyone knows are not ready for final release.

Collection and management of defect data may be (actually, should be in my opinion) more formalized during informal testing as described in the Chapter 9 discussion of defect management. (Earlier is better.) It is not uncommon for the vast majority of software defects found in a project to be found during this informal test activity that may start concurrent with the start of software implementation. Whatever can be done to reduce overhead during this period of time will allow the tests to be executed faster and the defects or other problems to surface sooner. That in itself will allow the development and validation effort to complete more efficiently.

### Formal Testing

Figure 13.9 shows the relationship between informal testing and formal testing. The top row of this figure represents individual test releases of software. Those releases to the left are considered informal test releases, and those to the right of the software readiness review considered to be formal test releases. Defect reports and defect management are always treated as formal activities, regardless of whether they were found during formal or informal testing.

Testing is represented in the figure in three groups: the verification tests, the ad hoc and session-based tests (SBT), and the validation tests. To the left of the test readiness review, each of these classifications of test is depicted as a continuum. In other words, it is not expected that a full complement of testing be completed on each of the informal software test releases. The objective during this informal pro-



**Figure 13.9** Informal versus formal testing.

cess is to maximize the time spent finding defects rather than being constrained by a rigid process that would require too much synchronization and analysis to draw conclusions that are not yet ready to be made. Activities should be focused on writing software, testing software, fixing defects in the software and tests, and verifying those fixes.

Formal testing, as defined here, should start when a version of software is released for test that everyone feels is possibly the version that could be the final released version. That is, there are no incomplete functions or features and no known defects that are still to be corrected. They are often referred to as candidate releases, and that designation is not taken lightly because of the cost and schedule implications. The first candidate release is so designated after a software readiness review. The purpose of that review is to go over the requirements and design documents to confirm that they are in agreement with the version of software being submitted as a candidate release. A further objective of this review is to confirm that all software requirements and design requirements are believed to be met in the software whose readiness for test is being reviewed.

This testing is considered formal because all test results from this point forward should be collected and organized to be included in the design history file for the product. Test results are reviewed for any questionable notes or marks that might indicate that the test completed with anything less than an uncontested pass (i.e., making it “objective evidence”). Any notes, marks, questions, or incomplete sections are analyzed by the results reviewer to determine the final disposition of the test. For example:

- The test passed, but there was a documentation defect (tester forgot to mark something).
- It was unclear if the test passed, it needs to be rerun.
- Tester’s comments researched, the test passed because \_\_\_\_\_.
- Defects were found and are documented in \_\_\_\_\_.

There could be any number of such dispositions; reviewing test results can take a considerable amount of time. The results of the reviews may get into the defect management process for determination of how the defects are to be handled. This involves other stakeholders, taking more resources. The point is that starting formal testing too early, that is, before the software has passed a rigorous readiness review, will result in delaying the completion of the testing and thus the product release. It is not uncommon for companies to feel that they are accelerating a schedule (or meeting a schedule) by starting formal testing before the software is ready. This almost always results in the schedule being further delayed because of the additional overhead premature formalization will impose.

Final, formal testing should progress quickly and smoothly, if informal testing ran in parallel with implementation, and if the readiness review qualified the software before formal testing starts. Ideally, there are no surprises during formal testing. The surprises were found and dealt with during informal testing.

Of course, that is the ideal situation that seldom if ever materializes. More likely, the formal test activities will be spread across several candidate software test releases. Referring back to Figure 13.9, the three rows representing the test activities

are represented as discrete versions of test results to the right of the software and test readiness reviews rather than the continuous band that represents the testing activity to the left of the reviews. Why? Each of the candidate software test releases is potentially the last test release, and consequently potentially the last round of testing. For that reason test management must carefully consider what is tested and what is not tested on each candidate release.

## Regression Testing

This leads us to the topic of regression testing and regression analysis. Technically, a regression test protocol is a collection of test procedures that is run on a software test release that tests for any unintended changes to the software's behavior that are unrelated to the specific changes made between test release versions. There also will be intentional changes in the software's behavior related to the intentional changes made between candidate versions. Those intentional changes are usually related to the resolution of a defect once in formal testing. Clearly, intentional changes should be directly tested in the first version in which they are made since once the software has changed, it becomes untested. It is common for companies to refer to both of these kinds of tests that are run between candidate versions of software as regression testing. Which way you use the term is not as important as understanding that the testing from version to version should include tests for both intentional and unintentional changes.

Selection of tests for intentional changes is straightforward. The test already exists and should be rerun, or the test does not exist and needs to be created before it is run. The selection of tests for the unintentional changes is less obvious; however there are some guidelines that can be used for facilitating the regression analysis.

1. Tests of safety critical functionality are prime candidates for a regression test suite. It makes good sense that if this is potentially the last round of testing, one would want to be sure the safety critical functionality is still working before the software is released.
2. Software in which there is a reduced level of confidence should also be considered as part of the regression test suite. How exactly do you determine if your confidence certain software functionality is reduced? Figure 13.10 is a simplified test management matrix to provide examples of how this determination might be made. Assume for now that none of these tests are for safety critical functionality since that would overrule any of the following analysis. In this matrix, the columns represent candidate test versions of the software, and the rows represent test procedures for specific software functionality. The cell at the intersection of the row and column contains the results of that test on that version of software. Patterns of test results can be used in a qualitative way to determine the level of confidence in the functionality that is tested by a given test.

Consider TST-01. It passed on version 0.00 of the software, but had not been rerun for the next eight versions of the software. There is nothing to indicate that that functionality is broken. However, a large window of time has given defects a lot of opportunity to creep into this functionality

	0.00	0.01	0.02	0.03	0.04	0.05	0.06	0.07	0.08
TST-01	P								
TST-02	P	P	P	P	P	P	P	P	P
TST-03	F	F	F	F	F	F	F	P	
	F	F	P	F	P	F	P	P	P
TST-05	F	P							
TST-06	P	P	P	P					
TST-07	-	-	-	-	-	-	-	-	P
TST-08	F	F	F	F	F	P	P	P	P
TST-09	P	-	-	-	-	F	P	P	P

Figure 13.10 Test result matrix.

undetected. TST-02 has been run and passed on every single version of software (for whatever reason). One's confidence in that functionality would be relatively high. TST-03 tests functionality that did not work correctly until version 0.07 of the software. That could be that a repair was not attempted until that version of software, but it could also mean that the fix was troublesome and didn't work until version 0.07. One's confidence level would depend on which of those situations is correct. TST-04 has had three iterations of failing and working. It looks like troublesome software functionality that should be tested frequently, and definitely before release of the software. TST-05 has not been repeated for seven versions of software and had as many failures as passes (one each). One might conclude that it is time to run it again. TST-06 is borderline on the confidence scale. It has no history of failure, but it hasn't been rerun for five versions. TST-07 was only tested once in the last version of software, but it was a pass, unless there is some direct or indirect connection to the changes made in the upcoming version 0.09, our confidence may be relatively high. TST-08 failed in the first five versions of software, but has been stable in the last four versions. Confidence in TST-08 is relatively high. TST-09 has passed on the last three versions, but it passed in version 0.00, then failed in version 0.05, so there is a history of defects resurfacing. Confidence is borderline.

This example is not to be interpreted as rules for regression analysis. Instead, it is meant as an example of the critical thinking that should go into the analysis. Further, it points out that the same data can be interpreted in radically different ways. One needs to dig deeper for the facts, or needs to err on the side of testing if in question to be on the safe side.

3. Software functionality that is indirectly related or indirectly traceable to an intentional change may be at high risk of having inherited a change that will make its behavior unexpected and defective. This is perhaps the most difficult analysis and sometimes requires broad knowledge of the software requirements and the software design. As an example, suppose a change is

made to a data entry field to allow the limits of acceptable numeric inputs to be extended to increase the range of allowable inputs. The direct test is the one that tests that the data entry field allows the appropriate new range of inputs. The tests of indirectly related functionality are those that use that variable, as in input to be sure that their functionality has not unintentionally been compromised with the expanded range of values of the variable.

## Automated Testing

Imagine pushing a button to rerun all of the tests for a device overnight. Regression analysis to pick a regression suite subset of the test procedures would not be necessary. It would be so fast and so easy, one would just run all of the tests.

That is part of the dream of automated testing. Automated testing can indeed be a powerful tool, but it can also be an expensive toy that may not deliver the test coverage of more traditional manual methods. Automated testing has its pros and cons like any other test tool. Let's examine both the pros and cons:

### Pros

1. Test execution is very fast, uses much less manpower, and is less expensive.
2. Test execution is very repeatable. It is not prone to boredom or human error.
3. Because of the speed and accuracy of automation, some tests are possible with automation that are not possible manually (stress and load testing, timing tests, communication response time testing, etc.)
4. Automated testing is excellent for multiplatform software, so that retesting on the second and succeeding platforms is automated with little test development effort. In particular, software that runs on PC or mobile device platforms are good targets for automated testing because the platforms, which often are from consumer supply channels, can have very short availability windows before they are replaced with the next version or generation device. Automated tests can greatly reduce the time and cost of retesting device software on a new platform. Of course, those advantages will not apply to software functionality that changes from platform to platform. In particular, user interface testing tends to require some revision across platforms to accommodate changing display technologies, display resolutions, and user input devices.

### Cons

1. Development of automated tests can be expensive. Depending on the tool, the tests may need to be designed, test cases selected, procedures outlined, before the procedures are developed in the tool. That test development step often looks like writing software, and takes trained, skilled developers. Aside from cost issues, this also makes it difficult to scale up a project quickly.

2. Maintenance of test procedures can be expensive. Several times in this text it has been mentioned how expensive it is to make changes late in a development life cycle. That is because the hierarchical tree of documents, software, and tests that depend on each other have to be maintained for each change. Adding automated tests to this list only adds to the maintenance expense. Some automated tests are of a complexity level that is similar to the software being tested, which further adds to the maintenance costs.
3. Tests may actually be less comprehensive than traditional manual tests. There is nothing about the tools that makes the tests less comprehensive, but experience has shown that test automation can be complex enough that attention is diverted from the logic behind the test to the test automation tool itself. Often test automation is not controlled by a process that requires design and review before automation, which only exacerbates the potential for lower quality tests. In fact, automated tests may be difficult to review depending on whether a human readable script is available or not. Finally, some of the user interface driven test automation tools can be very simple to use, but invite the test automation team to slip into an ad hoc style of testing rather than well thought out, reviewed designs before automating the tests. In other words, automation is not a replacement for good test engineering process.
4. Automated tests can actually be too sensitive sometimes. Overly sensitive tests can lead to many false positive results. That adds clutter to the results and introduces the possibility that a true defect could be overlooked. Excessively sensitive automated tests can result in extra work to review the test results to identify which test failures are real and which are false. If the test plan requires that the tests be modified and rerun to eliminate the false positive results, yet more overhead is required.
5. Automated tests are good at what they do, but they are not a guarantee of good tests. That is the responsibility of the test developer. Test automation tools are simply good at automatically running tests quickly. They will run both good tests and bad tests quickly.

## Summary

This chapter has covered a lot of territory. Levels of tests, test methodologies, test development processes, and even tips on where defects are likely to be found have been covered. There is so much information here that it may be worth skimming a second time to reinforce the topics and organization.

Testing is imperfect just as development is imperfect. However, that is no reason not continuously to try to improve the test process. Doing so will improve the results of testing and increase confidence in the resulting software through recognition of the completeness and thoroughness of the test effort.

## References

- [1] Weinberg, G. M., *The Psychology of Computer Programming*, Van Nostrand Reinhold Company, 1971 (1998 current edition).
- [2] AAMI TIR32:2004, Medical Device Software Risk Management, *AAMI Technical Information Report*, December 2004.
- [3] IEEE Standard 829–1998, *IEEE Standard for Software Test Documentation*, The Institute of Electrical and Electronics Engineers, Inc., 1998.

## Select Bibliography

- Beizer, B., *Black-Box Testing: Techniques for Functional Testing of Software and Systems*, New York: John Wiley & Sons, Inc., 1995.
- Hetzell, B., *The Complete Guide to Software Testing, Second Edition*, Wellesley, MA: QED Information Sciences, Inc., 1988.
- Jorgensen, P. C., *Software Testing: A Craftsman's Approach*, Boca Raton, FL: CRC Press LLC, 1995.
- Kaner, C., Bach, J., & Pettichord, B., *Lessons Learned in Software Testing: A Context-Driven Approach*, New York: John Wiley & Sons, Inc., 2002.
- Kaner, C. Falk, J. & Nguyen, H.Q., *Testing Computer Software, Second Edition*, New York: Van Nostrand Reinhold, 1993.
- Kit, Edward, *Software Testing in the Real World: Improving the Process*, Addison-Wesley Publishing Company, 1995.
- Perry, William E., & Rice, Randall W., *Surviving the Challenges of Software Testing: A People-Oriented Approach*, New York: Dorset House Publishing, 1997.

# The Maintenance Phase Validation Activities

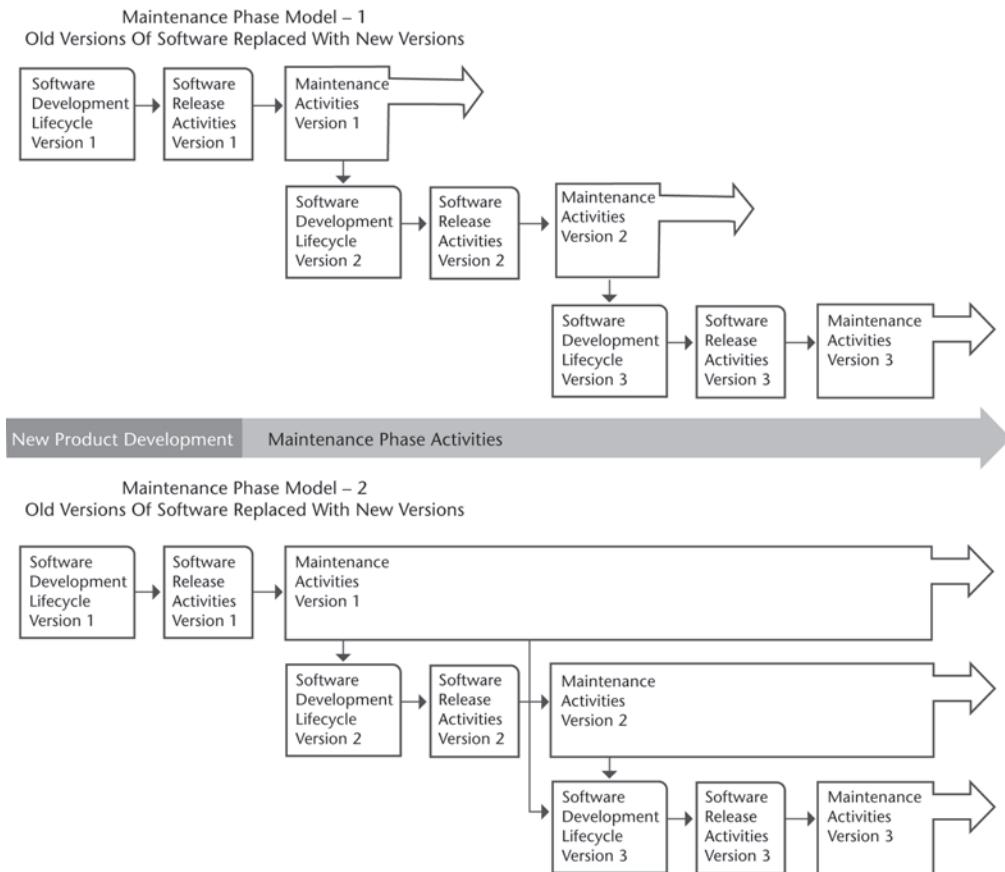
## Introduction

One might think that when the developers and testers agree that the device software is ready for release that the software is validated and the job is done. Those who think that way are in good company because many experienced engineers in the industry think, or at least act, that way. In fact, at this point, the device and its software are about to *begin* the longest phase of the life cycle—the maintenance phase. Recently, a well-known infusion pump was ordered by the FDA to be recalled from the market and destroyed after it had been in the clinical environment for about 13 years. Let's guess that device took two or three years to develop. Even with its truncated lifetime, the device spent roughly 85% of its total life cycle in the maintenance phase. From a business perspective, this is the phase in which the device manufacturer actually derives income from the device, rather than accumulating expense. There is good reason to keep a device in this phase as long as it is competitive in the marketplace to maximize the return on the development and validation investment. Keeping a device's software validated over this long time frame poses challenges unique to product maintainance.

A lot of things can happen in 13 years that can affect a device in the marketplace. Defects may be discovered by users, or feature changes may be needed to respond to market pressures. A lot of engineering budget will be consumed in maintaining a product over that period of time. It only makes good business sense that some planning should be in place for the maintenance phase, and to plan for the protection or improvement of the state of validation of the device software after its initial release.

What exactly is the maintenance phase? There is some ambiguity in what is meant by a maintenance phase. Figure 14.1 illustrates some of the ambiguity. In this figure, after the release of version 1 of the software, the maintenance activities begin. At some point in time, those maintenance activities lead to a conclusion that a version 2 of the software will be required, and a new development life cycle begins. That development program could take a number of months (or longer) to complete, so the maintenance activities of version 1 must continue during that time. When version 2 is complete, a decision must be made as to whether version 1 is retired from the market, or is allowed to exist contemporaneously with version 2. Both options are represented in the figure. The top model retires the old software from the field, the bottom model allows old versions to coexist with the new version.

Referring to the bottom model of the figure (both versions exist in the field), when version 2 is complete there are two parallel maintenance phases for the two



**Figure 14.1** Relationship of maintenance phase to development phase(s).

existing versions. Alternatively, one could consider it all one large maintenance phase. For the purpose of discussion, in this text “Maintenance Phase Activities” as shown in the band in the middle of Figure 13.1 will refer to all activities that take place after the *initial* release of the software/device, regardless of how many versions are supported in the field.

In Chapter 5 it was mentioned that attempting to fit every development project into a single software development life cycle model was problematic. Specifically, in the context of maintenance, the development life cycle for the new product development of the version 1 software is likely to look very different from the development life cycle for adding a few new features in version 2. Likewise, trying to fit every post. The first release development life cycle to a single model template is likely to be difficult. The details of what maintenance looks like and how the product evolves will be driven primarily by events that are impossible to predict and therefore impossible to plan in too much detail. A new release of software could be necessary within a few weeks of release if there is a critical defect found in the field. On the other hand, a stable product may last many years on the market without a critical update or routine software upgrade. What *can* be planned is the process of collecting the maintenance phase inputs, analyzing them, deciding which changes are made and when they are made, and keeping the product in a state of validation.

One should recognize that medical device development and long-term maintenance are not simple stenciled models that apply universally, and that there are ambiguities in the ways that the terminology is used. More importantly, one should focus on the underlying activities to be conducted and the decisions to be made in maintaining software.

Who is responsible for the maintenance phase activities? Validation activities in general overlap significantly with good engineering practices. In the maintenance phase activities, there is significant overlap not only with development practices and validation practices, but also with the testing group, quality assurance, regulatory compliance, and manufacturing. The boundaries of responsibility are not at all distinct.

Maintenance is also getting more difficult. Historically, when the device software was much smaller, less complex, and not as regulated as it is today, there was an attitude in the industry that the software was disposable. When the number of changes that were needed reached a certain threshold, the device would be redesigned with newer technology, and the software would be rewritten. Consequently, the maintenance activities may not have seemed so critical, and responsibility for those maintenance activities may have been delegated to a less experienced group as a learning opportunity for them. In today's reality, device software has become extremely complex, and devices may take years to develop. There is more invested in the software's development *and* validation than in years past. Replacement takes longer and is more expensive than in the past. Device software is now viewed as more of an investment or asset that manufacturers are less likely to throw away and replace. Rather, manufacturers are likely to extend the life of devices to maximize the return from the high development costs. Reuse of code is more common than in the past for the same reasons. Consequently, software is even more likely to be in a maintenance phase for longer than it has been historically.

As the maintenance phase for software has lengthened, the size and complexity have increased to the point that the implications of even minor changes in the software are not obvious and require a certain level of expertise with software maintenance and the device technology itself. In a best-case scenario, the documentation left behind by earlier generations of developers and testers is valuable in this endeavor. Frequently the maintenance team has to work with less than best-case situations. Together, the increased length of time in maintenance and the increased complexity of the maintenance task combine to increase the risk of software defects in the maintenance phase.

If one accepts that there is risk in the maintenance phase, then one should also appreciate that planning for maintenance could reduce that risk. In fact, designing the device for maintainability not only reduces the risk of future defects, but could also reduce the cost of maintenance. Much of the "design for maintainability" of the software has to do with the engineering development and validation documentation that is in place at product release. That implies that planning for the maintenance phase should actually be considered early in the design and development process.

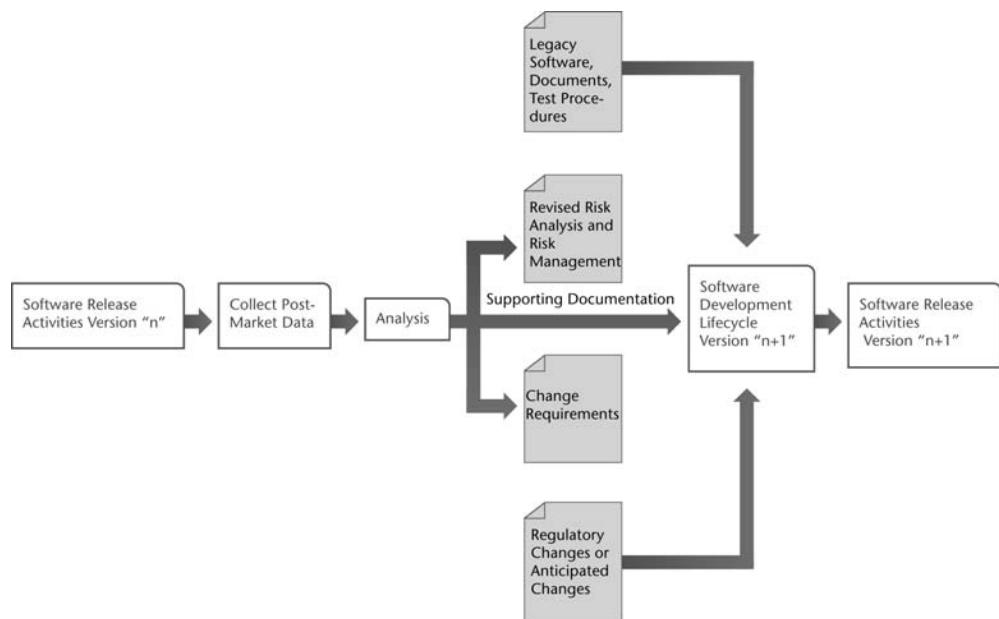
## A Model for Maintenance Activities

Figure 14.2 illustrates a maintenance phase model for validation-related activities that take place after the software-driven device completes final testing in the software (new product) development life cycle. This is a subset of the Maintenance Phase Activities referred to in the previous figure. There are many other activities other than the software engineering activities shown here such as, manufacturing, quality, regulatory, compliance, and other technical disciplines. This model only represents activities that are purely software validation activities or interface with validation activities.

The maintenance phase begins immediately after the new product software is released. However, since the software release activity was not discussed as part of the testing phase activities, it will be discussed here as an activity that transitions the software into the maintenance phase. The activities in the model repeat, or iterate, for as many postinitial releases of software that there are. We will refer to the activities related to any single iteration as the “maintenance phase activities” and the entire life cycle after the initial release of the software as the “Maintenance Phase” (capitalized).

Technically, the Maintenance Phase as proposed would begin with the collection of postmarket data, but let us start with the release activities to better illustrate the point that an iteration of maintenance phase activities spans from release to release by starting the figure with the release activities. Coincidentally, the release activities were not covered in Chapter 13, so they will be covered in some detail here.

Once a version  $n$  of software has been released to the market, data will become available on the performance of the product in the marketplace in its intended use environment. That data is collected and analyzed over some period of time until it is



**Figure 14.2** Model for maintenance activities.

decided that there is adequate need or urgency to begin the development of a new version of the software. (Again, we are only considering maintenance of the device software here, but obviously similar analyses and decision making would apply to other aspects of the device.)

A number of inputs are available to that maintenance phase software development process. Some of those inputs, such as the revised risk management report and change requirements are direct outputs of the maintenance phase analyses. Other inputs include the legacy software itself, any design artifacts, and any regulatory changes that might impact the development and validation effort. Depending on the urgency, depth, and extent of the changes to be made, the development life cycle of the software may take a variety of forms. For now, let us assume that the testing and validation activities are incorporated into the development life cycle. At the end of the development and validation activities, this iteration of maintenance activities concludes with the release activities for the new version,  $n + 1$ .

That is a brief description of the structure of the proposed Maintenance Phase for device software. To clarify, let us expand on each of those boxes in Figure 14.2 in some detail.

To better understand the Maintenance Phase, one should look at the components of a maintenance iteration in some detail. The following is a breakdown of some of the details of the components. A maintenance plan would further detail each component at each release iteration.

### **Software Release Activities: Version $n$**

Software release will be treated in this text as an activity that is part of the Maintenance Phase. It could also have been considered as a final phase of the development life cycle. Considering it as part of Maintenance allows us to generalize on what might go into the release activities for each release (initial release and maintenance release) of the software providing some consistency across the total lifetime of the software. In the context of this text, attention will be focused on verification and validation activities, recognizing that there may be many other maintenance activities in any organization.

#### **Verification of Completion of Testing and Evaluation of Results**

Presumably completion criteria were established in the software verification and validation plan or separate criteria were defined in the individual test plans. If this is the case, it is a simple matter to determine whether the criteria have been met. If not, this will be a more subjective process that should be directed more by safety considerations than budgets and schedules.

The risk-based completion criteria should include:

- Verification that all software implemented risk control measures are traceable from the risk management definition of the control measure to the test or tests that verify the correct implementation of that control measure.
- Verification that all tests that do trace back to risk control measures (i.e., safety critical tests) have been executed with acceptable results.

- Analysis of all defects in safety critical tests that are deferred for repair. This analysis should include verification that the deferred defects do not represent a possible cause for a hazard, or are of sufficiently low risk as to rationalize their deferral.
- Analysis of all defects in nonsafety-critical tests that are deferred for repair. This analysis should include verification that these failure modes do not represent new causes for hazards that were previously not predicted.
- Analysis of the “version spread” for the entire collection of tests, especially the safety critical tests. This analysis gets tricky when a number of generations of regression tests were compounded for the set of final test results. Verify the validity of test results for other than the final release version. Investigate whether any changes were made in the general functional “vicinity” of the safety-critical code that might suggest reexecution of the tests for that code, especially if it is safety-critical code.

#### Verify Agreement of Software and Documentation Versions

At the time of release, the state of the entire documentation tree from hazard analysis to test procedure should agree with the implemented version of the released software. One should not be tempted to release the code to production without updating the documentation to reflect any requirements changes or corrections, design changes or corrections, test procedure corrections, or any additions or corrections to the risk analysis/management. Failure to bring the documentation into agreement with the released code will mislead the development and test team on future releases of the product, and could lead to overlooking hazards and their causes, or their “implied” control measures that are not fully documented.

#### Documentation of Procedure and Environment Used to Build the Software

The objective in this task is to leave enough documentation behind that the next generation of developers can recreate the executable code, byte for byte, from the source code. This re-creation of the previous release should be the baseline for any future development. It is an important verification task to be sure that the executable code can be re-created from the source.

This verification is important because any unexpected changes to the software that are introduced by slightly different build procedures or different versions of software libraries represent potential unexpected causes for hazards in the re-built software. Any assumptions about test results from prior versions are not valid if changes to the software cannot be controlled because of changes in the build procedure, software versions, or development environment.

This task seems simple to the inexperienced, but experience has proven that many projects struggle to re-create the executable code at the end of a project. Imagine how difficult it would be if the next iteration took place several years later.

One should beware of hidden environment settings that would keep the software from being built on a newly created environment. A good practice is to test build procedures on a totally separate system before archiving the instructions away.

The version numbers of any tools used to develop the software should be documented, especially compilers. One can no longer go by the version on the label on the box or release media that was originally purchased. Internet-based automatic updates to tools and operating systems make it difficult to know, let alone document, the versions of tools and operating system components. This problem has become complex enough that it is becoming increasingly common for device manufacturers to “freeze” an entire development station at the end of a project to guarantee that they can have an identical starting point for future development work on a product.

Documentation of the environment is not limited to software development tools. The environment includes any tools that were used for the development of any of the documentation such as requirements management tools, static analysis tools, defect tracking systems, version control system (VCS) software, and even word processors, spreadsheet software, macros and any other software that may be needed to access documents related to the product. One should be especially careful of add-on products such as requirements trace tools that work within the word processor and database. Compatibility issues abound in these situations. A loss of documentation and related trace links could lead to lost risk related documentation.

#### Archive and Maintain Master Copies of Executable Software, Source, and Documentation for Life of Product

Archiving the source code and documentation files is clearly important for obvious reasons. To make use of the source and document files in the future, however, one will need copies of the tools that were used to create these files. It is good practice to archive copies of the tools along with the source and documents. One should be especially careful to archive any off-the-shelf (OTS) packages that are embedded into the product, and libraries that may be packaged with the compiler, assembler, or linker.

A good archiving policy is good business sense, but also relates to risk management in the developed device. Any inability to re-create baseline software, documents, or trace links on a future release of the product will introduce a possibility for an undetected change into the product. Assumptions about the safety of the baseline product based on past testing or field experience are invalid if one cannot recreate the released version of software.

The long list of potential tools that make up the development environment should also be archived. It does absolutely no good to document the version of the tool that was used to create the release baseline of software if the tool cannot be found years later when changes are needed. One should be especially careful to archive version control system (VCS) software and the data files for the project. There is a good probability that future VCS releases will not be able to read today’s data files.

The life of the product might well be longer than the life of the current archive media. For example, imagine that your only archive copies of source code were on 8-inch floppy disks! Not only is obsolescence a problem with media, some media such as tapes, CDs, and DVDs deteriorate over time. (We recently moved data from CDROM archives that were about 8 years old and had read errors on about half of

the disks! Luckily, our plan called for duplicate archival copies stored off site and we were able to recover all data.) Part of a maintenance plan should include verification of the integrity of the archive information as well as periodic review of the availability and viability of the archive media. A second set of off-site archive copies is a good practice (and is what saved us from the deteriorated CDROMs).

#### Label, Package, and Deliver Software in Accordance with Established Procedures

Devices with visible software driven user interfaces should have a means to display an identifiable version number at power up or through a simple user selection. It should be verified that the same version number is displayed or labeled onto whatever medium the software is delivered on (DVD, CDROM, flash drive, etc.) or is used by production for adding to programmable parts. A user should not be required to work too hard to determine what version software is in his or her device. This is especially important for software safety if dangerous defects are identified after the product is released to the field.

A summary “tested configuration list” should be in place to identify which versions of subsystem software and hardware components were tested together as part of the entire device’s systematic testing process.

The versions of any other firmware in other programmable parts in the device configuration that do not have the potential to be displayed on the user interface should be clearly labeled on the device, and optionally in an electronically readable form to allow for automatic identification, test, and lockout of invalid versions.

Once it has been determined that software testing completed satisfactorily, and the above details have been verified, the software is ready for release and enters into the first maintenance phase iteration.

### Collection of Post-Market Data

Shortly after the device and the device software are released to the market, new data becomes available that are important inputs to the maintenance activities for the device. Information can be collected actively or passively.

Passive post market data is unsolicited data that comes to the manufacturer from customers, users, sales people, service people, and others. Oftentimes, the passive data comes in the form of a complaint or suggestion. There are a number of regulations and guidance documents related to the handling of complaints that are beyond the scope of this text. However, it is worthy of note at least that some forms of passive post market data result in manufacturer submitted medical device reports (MDRs) that have visibility at the FDA.

Actively gathered post market data does not come to the manufacturer without manufacturer initiated actions to collect it. This distinction between actively and passively collected data is not critical except to recognize that there are opportunities, perhaps even obligations, for the device manufacturer to actively seek out information about the device, its usability, and its acceptability in the marketplace.

## Process and Planning

As important as collecting post market data is managing that data in a way that assures the continued safe operation of the device in the market, and manages the improvements made to the device to address market needs. A maintenance plan needs to implement a process for managing the collection of post market data, the analysis of that data, and the response to that data. The 62304 software life cycle standard [1] requires a maintenance process and plan implementing that process that addresses:

- Receiving feedback data (i.e., post market data);
- Documenting feedback data;
- Evaluating feedback data;
- Resolving issues raised by feedback data;
- Tracking the resolutions of issues.

## Sources of Post-Market Data

Let us take a look at a list of sources for post market data. This list is not meant to be a comprehensive list, but is meant to serve as an example to stimulate thinking about what sources of information would be appropriate for a given company, device, or project.

*Complaints.* This is rather self-explanatory, and has regulatory overtones, but it is important to note that complaints should not be thought of only as passively collected data. It is quite common that when users are *asked* if they have experienced any problems, they often report problems that have never been submitted through normal written channels. An even different list of problems and complaints can arise from simply *observing* how users use the device in a post market clinical setting.

Several years ago, we interviewed users of a device that had been in the market about 5 years. The purpose of our user interviews was to solicit information that could be used to improve the next generation of the device; however, a number of complaints were articulated that our client had never heard about before. Even more surprising, when our team members observed the use of the device in the clinical setting, they noted several system crashes that required rebooting the PC upon which the device software ran. It turns out it was a fairly widespread problem, but nobody thought to report it because rebooting a PC seemed “almost normal” to the users because they had developed a high tolerance to rebooting from using their office computers which used the same operating system.

*New feature or function requests.* These too may be collected passively from users, sales people, or other interested stakeholders. They can also actively be collected by communicating with users and other stakeholders during the Maintenance Phase of the product. Given that users do not always communicate their complaints to the manufacturer, they are probably less likely to initiate communication of their ideas for new features or functions that would improve the device. The best sources of improvement data are actively generated.

*New opportunities.* New opportunities for the device may stem from new therapies or changes to existing therapies that may require some modification of the device. The same is true of diagnostic devices; changes in the diagnostic algorithms or analysis of results may create incentive for making corresponding changes to the device or the device software.

*Changes to external interface.* If the device communicates with some external device or network, anticipated changes to that communication interface could require changes to the device software.

*Performance improvements.* This is really a subcategory under feature changes or new opportunities, but users may be totally satisfied with the device but would like more throughput or would like for it to respond faster (i.e., would like for it to run faster). Alternatively, it may be that the device meets all of its claims related to accuracy, but once in the marketplace users for whatever reason decide that more accuracy would be better.

*Regulatory change.* Regulatory changes may necessitate changes to a device based on new guidance from the regulatory agency. Changes in regulation may also impact how a device is designed, documented, or validated. For example, consider the impact of a change to the FDA's design control regulations. Although this kind of change may not necessitate immediate changes to the device, it would certainly be an input to be considered when changes are necessitated for other reasons.

*Technology changes.* Obsolescence of technology embedded in a device certainly would necessitate a design iteration of the device. Improvements in the technologies embedded in a device also are inputs to be considered in determining whether a new design iteration is needed. Note that technological improvements may be implemented either in the hardware or the software, but even hardware improvements are likely to lead to software change.

*User groups and user web sites.* These resources are sometimes available for devices that are widely used. Unfortunately, they sometimes spontaneously form in reaction to a problem or series of problems that surface in the market. They also represent an opportunity for manufacturers to organize such gatherings or depositories of user information. Either way, they are rich sources of information that should be considered in Maintenance Phase analyses.

*Competitive information.* This may come in the form of information on new products or features offered by competitors that may necessitate a response in a maintenance release of the device or device software. Alternatively, this information could come from watching competitors' web sites, their user web sites, their user group publications, or public regulatory information related to problems (device or quality system) a competitor is having. Information about competitive problems should be included in subsequent analysis to determine whether similar problems exist or have the potential for materializing in the device being maintained.

The above list of data sources is a good starting point for where, what, and how a manufacturer should go about collecting post market data on a device in the market. Hopefully it stimulates some creative thinking on the part of the reader to come up with other sources of information that may be relevant generally, or to the specifics of a particular device.

It is likely that a large volume of data could be involved. Planning is necessary not only to ensure that the data is collected, but also to establish how the data will be saved, distributed, and used to make decisions related to future releases of the software. The 3Rs of planning (roles, responsibilities, resources) certainly apply to planning the Maintenance Phase as much as they did to development and validation activities.

## Analysis

There is a reason that all of the post market data is collected. It must be analyzed for determining which changes will be made to the product, in what priority order the changes will be made, and when the changes are critical enough, or simply numerous enough to initiate a development project for a new release of the software.

The IEEE standard on software maintenance [2] suggests that software changes be categorized into one of the following:

- *Emergency changes*—urgent changes that are needed regardless of the planned release schedule;
- *Corrective changes*—those made to correct known defects;
- *Adaptive changes*—those made to keep the software current in an environment of changing user needs;
- *Perfective changes*—those made to improve the performance, maintainability, accuracy, or cosmetic acceptability.

Prioritization of which changes go into a given release should be based on several considerations, one of which might be which class of change from the above list applies to the proposed changes. It is clear, for example, that emergency changes would be a much higher priority than perfective changes. It probably is not practical for one to prioritize strictly on classification boundaries. For example, implementing all corrective changes in a release, but no other category of change (obviously assuming there are no emergency changes pending).

The results of this analysis should be documented and used as inputs to the software development life cycle that will follow, should the analysis lead to the conclusion that a new release of software is required.

Those inputs to a subsequent development life cycle include:

**Change requirements.** A change requirements specification is an itemization of the changes that are to be made in the next release of software. These change requirements should be traceable to the specific changes that ultimately will be made in the software requirements, designs, implementation, and test procedures. For iterative releases in the Maintenance Phase, the change requirements are much

like the user needs in the product design and development project. These are the needs that justify the entire development project for the next release. Consequently, all of the validation activities that relate to user needs analysis and system requirements would apply to the change requirements.

*Revised risk management documents.* Part of the analysis of the maintenance phase inputs is a formal analysis and management plan for the risks associated with any of the inputs to the maintenance phase analysis process. Specifically, the risk of *not* implementing a suggested change should be evaluated, as well as the risk of implementing the suggested change. Some changes may be riskier than others. A change to a highly complex algorithm carries a risk that unintended consequences may result from that change. Changes that affect the balance of false positive/false negative rates may impose new risks. Any new risks that can be identified from the analysis inputs should be documented along with any possible risk control measures. Those risk control measures may apply prior to the response to the change request or after the changes have been implemented. The control measures may actually be part of the changes that are requested.

Regardless, the risk analysis should be one of the key ingredients of the maintenance phase analysis leading to the conclusion of whether or not to respond to a risk in a specific maintenance iteration. The details of that analysis and any identifiable risk control measures should be provided as input information to the subsequent software development life cycle, and should also be traceable into specific software requirements, designs, implementations, and test procedures.

Risk analysis should also consider the intended use of the device, and whether that intended use has changed since the device was first released or since the intended use was last revised. It is quite possible that intended use can drift once the device is released to the market. Intended use may change as new features are added to an existing device. Users can be creative in their use of devices, and manufacturers must be vigilant in collecting information on how devices are actually used. Most of this kind of information is collected actively, and most of that by observation. After all, how would a user know what the manufacturer's assumptions were about usage? Any change in intended use may introduce new hazards that had not previously been considered, or may violate assumptions about how the device was intended to be used that would have kept the risks at acceptable levels. Just as risk analysis and management span the entire development life cycle as new information evolves, the same is true in the Maintenance Phase. As more is learned about the device in the marketplace, risk continually needs to be reexamined.

*Supporting documentation.* The outcomes of any analysis performed should be shared with the development and validation team charged with implementing the changes outlined in the change requirements. The following is a list of information that may be provided to those responsible for implementing the changes, and implies that these topics should be considered as part of the analysis:

- *Applicability:* Software often exists in a multiplicity of configurations. For example, those configurations may have to do with the platform on which the

software runs or may be based on which feature set is enabled. The analysis should consider to which configurations the proposed changes apply.

For software that has been in the market for a number of years, there could be a number of versions of software still in circulation. The analysis of the proposed change should consider to which of those versions the proposed change would apply if a recall for upgrade is necessary. Further, it should be considered and articulated to the implementation team whether the proposed changes are meant to replace existing software in circulation or are simply meant to replace the software in newly manufactured devices, leaving the old software in circulation. All of these decisions are valuable inputs to the implementation and validation teams.

- *Impact analysis:* The impact of a proposed change should be considered on the real and perceived safety of the device, on the user (safety, convenience, desirability, training, etc.), on sales, and on serviceability. Any anticipated negative impacts should be documented so that the implementation team can minimize the impact in the design, and so that the validation team can ultimately test for the existence and magnitude of any resulting negative impact.
- *Alternative approaches considered:* Part of the analysis will certainly consider alternative approaches to any changes recommended. This deliberation should be documented and provided to the implementers not only to save them from having to repeat the same analysis, but also as a cross-check for whether the recommended approach was based on sound assumptions and data. If the alternative was selected because of an anticipated advantage over other approaches, the validation team might want to verify that advantage was in fact actually achieved.
- *History of prior changes:* It has certainly been repeated enough in this text that there is a higher likelihood of repeating an error than there is in making the error the first time. For that reason, the analysis should consider the history of prior changes to the area of functionality or software to which a recommended change is to be made. If an area of the software has been very error-prone, providing that history to the implementer puts him or her on high alert. Providing that history to the validation team allows them to focus more test effort on troublesome areas and to be certain that their test procedures include tests to assure that old defects have not been reincarnated as part of the change process.

The analysis of inputs for identifying changes in maintenance releases of software can become quite complex. The makeup of the team handling the analysis from iteration to iteration has a high likelihood of changing over time. Documentation of the analysis thought process and decision rationale for changes recommended and not recommended are very important for passing responsibility onto future generations. Analysis of input data seldom should be discarded and should be part of an ongoing evolving file of information to be considered for each new release of the software.

Maintenance phase analysis shares many similarities with change control and defect management that take place as part of the initial development process and

that were outlined earlier in this book. The principles are the same, but the context is very different. Defect reports take on a new level of urgency when the potentially defective software is already in use clinically. Management of defect reports, complaints, and other inputs are also more visible to regulatory oversight than routine defect management that takes place during development. Well-defined processes that describe how a manufacturer will manage, analyze, and respond to post market inputs is the best way to assure the maintenance team consistently manages Maintenance Phase activities, even as the team members change responsibilities over time.

## The Maintenance Software Development Life Cycle(s)

Once the Maintenance Phase analysis comes to the conclusion that a new release of device software is necessary, the inputs for that development life cycle need to be assembled and transferred to both the development and validation teams. Most of those inputs are produced by the analysis just discussed. However, there are at least two other items that, while no less important, are not an outcome of the analysis:

1. *Legacy software, documentation, and test procedures.* In a perfect engineering utopia, a prior release of software would have been thoroughly documented, traced, and tested. The behavior expected in the test procedures would actually be the behavior that was implemented and was documented in the requirements. In other words, there would be alignment in the “configuration” of the software, documentation, and test results. Unfortunately, this utopia is seldom the case, especially with legacy software that is more than several years old, or that has been patched and improved through several release iterations, or software that was inherited through merger or acquisition.

Enough has been said above under the topic of release activities about the importance of archiving and documenting the procedure for building the executable code from the source code. One common mistake that companies make in the rush to release the software is to cut corners by not bringing the documentation into agreement with the version of software that was ultimately released. Sometimes this need is simply ignored, other times redlined copies or a to-do list are filed away to be updated, if and when the software is ever updated. (Yes, there really are companies that believe that the software will *never* need to be updated again.) Although this may seem like a good idea at the time, it is guaranteed that it will seem less so when the files are opened up several years later by engineers who had nothing to do with the prior release of software. Cryptic notes no longer have the meaning they once did and those details that nobody will ever forget—well, they are forgotten.

When corners are cut during release, the next iteration of development is faced with the cleanup work from the prior generation before work can actually begin on the change requirements for the next release. That cleanup work is never desirable, and again some will further rationalize putting off the work of aligning software, documentation, and procedures opting

instead to use the change requirements as a mini-SRS. That does achieve getting the engineers to modify source code faster, but now the documentation to be understood for the next iteration includes the original, partially incorrect documentation, and the new change documentation. Cumbersome? Yes, but manageable, some say.

Let us think one move farther ahead to the third iteration of changes to be made in the Maintenance Phase. If version 2 took the “change documentation” approach, then version 3 almost certainly will take the same approach to avoid the now doubly messy documentation cleanup that is now spread across two versions of software and the change requirements for the upcoming release. Too often, companies decide it is just too complicated to fix it and they press forward with fragmented documentation.

Why does it matter? Assuming that the engineers and validation test team use the documentation as a tool, they will find that this fragmented documentation provides them with a tool that sometimes works because the documentation is correct, and sometimes it does not. Time is wasted by the developers in trying to understand why the source code does not agree with design documentation and requirements, or test procedures—and trying to understand which is the correct behavior. Testers will waste time because test procedures may no longer expect the correct behavior of the software because the tests are based on the requirements but the software began deviating from the requirements two versions ago. The bottom line is that almost always one will end up spending more time working around the confusion that was created by not cleaning up the software and documentation configuration during the release activities than was saved by cutting corners—unless you are one of those that believes the software will never have to change again.

If the additional cost of making changes in the future is not adequate incentive to clean up the configuration at the end of a project, then think of the hundreds of thousands of dollars to millions of dollars that were invested in the documentation. Why would it make sense to invest all that, and not harvest the benefit from it in future iterations? Think of the documentation as a million dollar airplane. Would it make sense not to apply the latest service bulletin updates to the plane, or not to maintain the engine? The end result is that future use of the plane (or software documentation) is likely not to be a pleasant experience. Investment in assets like these should also consider the cost of maintaining the asset.

2. *Regulatory changes.* It is rare that software would have to be modified simply to meet a change in the regulatory environment. When regulatory changes are made, or expectations evolve, they are, generally speaking, applied to device/software releases subsequent to the regulatory change.

Software development teams, and even quality and validation teams, are sometimes unaware of changes in the regulatory environment. Not communicating those changes before launching a development/validation life cycle could result in rework to comply with the changes or run the risk of some regulatory action.

Regulatory changes seldom happen overnight. They are often discussed in the trade media, industry workgroups, hearings, and tradeshow workshops far in advance of any required changes. Given that some development/validation life cycles run for months or even years, communicating anticipated changes in the regulatory environment to the development and validation teams allows them to anticipate the changes and take appropriate action to avoid a major change late in the project.

### **Software Development and Validation Activities**

Since two chapters already been dedicated to development life cycles and validation life cycles, there is little need to repeat the life cycle details here. In short, almost everything that applied to the original release of the product and software would apply to a Maintenance Phase iterative release. However, the sequencing and coordination of activities in the maintenance release life cycles may look quite different from those in the original release.

The activities themselves also may be slightly different, and this is why in earlier chapters it was suggested that the plans play an important role in software validation. Plans are written on a project by project (i.e., release by release) basis and can be tuned to the specific needs and circumstances of the project. Processes and procedures, on the other hand, are written to apply to a wide range of, if not all, projects.

We will not even attempt to go into all of those differences in much detail here. To do so would only mislead the reader into thinking that those differences would apply to all projects, when in reality one needs to think critically about needs and circumstances and plan a life cycle and the life cycle activities accordingly.

Let us examine the differences in just two of the activities as examples of how iterative maintenance life cycle activities differ from new product life cycle activities. The reader should not be misled into thinking that these are the only differences that exist between iterative and new product life cycle activities. One always needs to think critically about needs and circumstances and plan the life cycle and the life cycle activities accordingly. The two activities that are used as an example here are testing (specifically regression testing), and traceability.

When writing a test plan for a maintenance release of software, one might consider being more conservative in the regression testing plans if it is decided not to reexecute all test procedures. The regression analysis as mentioned in Chapter 13 should take into consideration functionality indirectly related to changes that are made in the software. (Of course, this assumes that all changes made during a maintenance release would be directly tested.) Why might one be more conservative in regression test planning on a maintenance release? One needs to be careful, thoughtful, and suspicious simply because more time has gone by. The contributors have changed, and there is a likelihood that the intricate interrelationships in the software that *may have been* well understood during the initial release of the software have now been forgotten. A good rule of thumb is, “when in doubt, test.” All too often test managers demand man-weeks of analysis and man-days of debate about how to reduce regression testing by several man-weeks. It seems that the time might have been better spent actually testing rather than debating how not to test.

The second activity used here as an example of how an activity's emphasis changes between initial release and a maintenance release is that of traceability. As mentioned earlier in this text, the traceability is an important tool for both the developer and the validation professional. Once a product is released to the market, design controls typically are tightened up. In other words, changes to be made to the software in a released device should be the subject of the maintenance phase analysis discussed above and a great deal of control over what changes are implemented and how they are verified. Traceability during new product development is primarily concerned with verifying that all requirements are actually implemented and verified through testing. Secondly, the trace analysis is used to verify that all code that is implemented is related to some requirement and design element in the documentation. That is important because that is the only way the previously undocumented functionality gets tested. In a maintenance release scenario, using traceability to detect unapproved changes becomes almost as important as tracing to be sure the change requirements are implemented. The main reason for this is for the test team to have a clear understanding of what *actually* has changed in the software so that they can test it and be sure that *only* those things that were supposed to change have changed. It is quite common for software developers to make perfective, but unapproved, changes to software in the same source modules as the software whose changes are approved. They do this to be helpful, because they notice an improvement that can be made. However, the unapproved changes bypassed the Maintenance Phase analysis that was carefully applied to all other changes in the change requirements specification. Using traceability to identify such unapproved changes calls them to the attention of the analysis team. If they are subsequently approved, the trace can be utilized by the test team to design appropriate verification test designs.

## Software Release Activities: Version $n + 1$

Once the software development and validation life cycles have run their course, the cycle repeats itself. The release activities for the  $n+1$  version of the software take place, and post-release data are gathered starting the next Maintenance Phase iteration.

This discussion of the Maintenance Phase activities concludes this section of the book that relates to medical device software validation. In the next section, validation of the software used to create, produce, or maintain the quality of the device is covered. Many of the principles discussed in this section will apply to the software that is the subject of the next section, but in very different ways.

## References

- [1] ANSI/AAMI/IEC 62304:2006, *Medical Device Software—Software Life Cycle Processes*, June 8, 2006.
- [2] IEEE Std 1219-1998, *IEEE Standard for Software Maintenance*.



**PART III**

## Validation of Nondevice Software



# Validating Automated Process Software: Background

## Introduction

There is no question that the validation of medical device software is important. There are, after all, documented examples of software failures that injured patients or that ultimately resulted in patient death. The pathway from software to device to patient is direct and easily understood. Consequently the pathway from a software failure to device hazard to patient harm is equally well understood.

There is other software that must be considered for validation, because it too can result in harm to patients and users, but in less direct ways. Software that is *used* by the manufacturer of a medical device could potentially fail in a way that could, although indirectly, result in harm to a patient or other user. The connections between this software and harm to a patient are less direct and somewhat more difficult to understand. However, that doesn't make the harm any less severe when a patient ultimately is harmed.

This chapter will break down the latter types of software that are covered by the FDA regulations and guidance documents. That is, the software that must be validated by law. That is, the validation for which documented objective evidence must be provided to regulatory agencies.

However, referring back to the opening words at the beginning of this book: "I don't trust software." Neither should you the reader, your employer, the government, or anyone else for that matter. Why we would ever want to exclude any software from some type of validation? The only reasons I can come up with are that the software is used for something so unimportant that we do not mind if it fails, or we trust that the supplier of the software used such well-controlled engineering development processes and has so thoroughly tested the software that further validation activity would be useless.

If indeed those are the only two reasons, then two more questions come to mind:

1. Why are you spending time and money on software to automate something of little or no value?
2. If the suppliers are doing such a good job developing and testing the software, why is it so common to notice defects in newly acquired software within minutes of its first use—even during its installation? The engineering process and testing did nothing to prevent even an installation error? Maybe the supplier never anticipated that one of the intended uses would be that it would be installed?

Am I cynical about software? You bet I am. One *trusts* that the supplier used best engineering processes and tested the end result so that end users would have no problems. If one cares about the process that software is used to automate, then *trust* should not be one's first line of defense when the software doesn't work as expected. Ronald Reagan made famous the quotation, "Trust ... but verify" (not in reference to software). The same applies to software—almost all software—trust but verify.

Once one starts to think about software in this way, one cannot help but question whether any software that is used (regulated or not) will really work for the purposes intended. Hopefully, that thinking will lead one naturally to think about the risks one takes with trusting so much to software.

What can be done to control those risks, and how one can build enough confidence in the software to trust it with valuable tasks or information? It is inevitable that in our digital world, whether one is aware of it or not, software will have to be trusted to control everything from our automobile engines and brakes, to our checking accounts, to the security of our credit cards—and to the safety and efficacy of our medical devices. In the medical device industry, software will ultimately have to be trusted to control production machine tools, to sterilize devices, to design parts, and to manage the critical information that is needed to produce the devices and retrieve them if they are defective. How does one know that the software works? How can one ever build the confidence to trust the software with these responsibilities for keeping the end users safe? There is no 100% guarantee with software. It always comes down to confidence and trust, but hopefully only trust after we have verified (and validated).

## Regulatory Background

The U.S. regulations for validation of nondevice software, or as they are called in Section 6 of the GPSV, Automated Process Equipment and Quality System Software, is regulated by two regulations, and is further treated in Section 6 of the GPSV.

As with other regulations that have been quoted in this text, these two regulations are short in their length, but very broad and far-reaching in their scope. The two regulations are

1. 21 CFR 820.70 (i) on Automated Processes;
2. 21 CFR 11 (Part 11) on Electronic Records and Electronic Signatures (ERES).

Before digging into the details of the validation called for by these two regulations, let us look at the specifics of what the regulations say, and understand what the words in those regulations mean in the context of software validation.

### 21 CFR 820.70 (i) on Automated Processes

The first of the regulations is under Production and Process Controls, 21 CFR 820.70 (i) Automated Processes:

When computers or automated data processing systems are used as part of production or the quality system, the manufacturer shall validate computer software for its intended use according to an established protocol.

—21 CFR 820.70 (i)—*Automated Processes*

“Intended use” has been discussed earlier in the context of the medical device, but in this context, it is the intended use of the software tool that is of interest. Validation for intended use implies that software tools do not need to be validated for any foreseeable use into the future, just the use in the context of the manufacturer’s requirements for the software and plans for how it will be used. That is an important concept. The device manufacturer does not need to validate for all conceivable uses, just the intended use.

“Established protocol” means the same thing here as it did under validation of device software: the validation is planned and documented. Poking around for a few minutes with the new software is not what the FDA has in mind, nor would that provide much value in building confidence in the software, but some don’t even do that!

It is clear that this applies to all software used in the production of the device, from software embedded in production machine tools to software that maintains bills of materials, production drawings and records, even software that makes process calculations and records production yields and other production data.

What exactly does that “as part of ... the quality system” mean? We all “kind of know,” right? The same set of quality system regulations provides a definition of quality system:

- (v) Quality system means the organizational structure, responsibilities, procedures, processes, and resources for implementing quality management.

—21 CFR 820.3 *Definitions*

So the automated process regulation applies to software that automates any part of the “... procedures, processes and resources for implementing quality management.” Does that help? It seems to have just pushed the question to a definition of “quality management,” which is not included in the definitions.

Luckily, in the context of software validation, the FDA guidance on software validation (the GPSV that has been mentioned so many times) provides some insight into what kinds of software might be considered as implementing quality management.

This requirement applies to any software used to automate device design, testing, component acceptance, manufacturing, labeling, packaging, distribution, complaint handling, or to automate any other aspect of the quality system.

—*General Principles of Software Validation: Section 2.4, January 2002*

At least in the opinion of this guidance, quality management includes not only production, but also design, development, testing, shipping, complaint han-

dling—and “any other aspect of the quality system.” This is not a regulation limited to software used in production, and by the quality control and quality assurance folks. This can be interpreted as software that in any way may affect the quality of the medical device. It is a regulation that is *very* broad in scope.

## 21 CFR 11 (Part 11)—Electronic Records and Electronic Signatures

The second regulation is 21 CFR 11, otherwise referred to as Part 11, which regulates electronic records and electronic signatures (ERES). The following is from the regulations for closed systems; that is, secured systems not available to the public:

### § 11.10 Controls for closed systems.

Persons who use closed systems to create, modify, maintain, or transmit electronic records shall employ procedures and controls designed to ensure the authenticity, integrity, and, when appropriate, the confidentiality of electronic records, and to ensure that the signer cannot readily repudiate the signed record as not genuine. Such procedures and controls shall include the following:

- (a) Validation of systems to ensure accuracy, reliability, consistent intended performance, and the ability to discern invalid or altered records....

—21 CFR 11—*Electronic Records; Electronic Signatures*

This regulation looks like other validation regulations that apply to medical device software. There is a subtle difference between this validation regulation and even the 820.70(i) regulation above. The mention of “validation to ensure....consistent intended performance” is slightly different in the context of Part 11 because the Part 11 regulation goes on to itemize 36 requirements (some compound) for electronic records and electronic signatures systems that define at least part of the intended performance. Of course, the user of the ERES system may have additional needs and intended uses, and may demand additional or more rigorous performance from the system, but the regulation establishes a minimum acceptable set of requirements for the ERES software that must be validated. That is different from any of the other validation requirements, all of which refer to intended use without prescribing what intended use includes.

What is different about Part 11 and validation is that the ERES software controls information (records and signatures) that the FDA uses and relies upon. In other words, the FDA becomes one of the users of the software and a direct stakeholder in the accuracy and reliability of that software to produce accurate records and reliable signatures (evidence of approval). The FDA does *not* have a direct stakeholder interest in device software, or in the nondevice software to which the other validation and verification regulations apply.

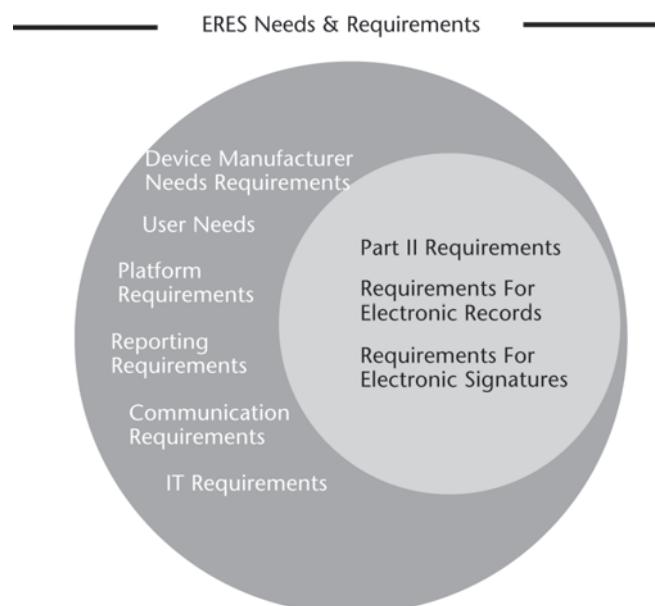
Is Part 11 regulated ERES software validation different from 820.70(i) Automated Process software validation? This question comes up frequently. Most reasonable software validation professionals would come to the conclusion that software that controls configuration management of documents and their electronically signed approvals could have an impact on the quality of the medical device and

therefore would be subject to the validation regulation of 21 CFR 820.70(i). But that same ERES software also has a regulatory requirement for validation in Part 11. So are they separate validations? Are they somehow different?

To clarify the situation, consider that verification establishes that specific requirements have been fulfilled, and validation is the full set of activities that builds confidence that user needs have been met. The diagram of Figure 15.1 shows the relationship of the device manufacturer's needs and requirements for an ERES system compared to that of the FDA as regulated by Part 11. The Part 11 requirements are those 36 requirements referred to above (and provided on the accompanying DVD). However the manufacturer also may have other needs and requirements of the software, in addition to the Part 11 requirements. The Part 11 requirements are a *subset* of the device manufacturer's requirements for two reasons:

1. Part 11 requires features and levels of performance that the device manufacturer would logically cite as functional needs and requirements for the device manufacturer's intended use of the ERES system.
2. Part 11 is a regulatory requirement, not a guidance recommendation. That makes it a requirement of the device manufacturer, thus making it a requirement of the software.

Therefore, since the Part 11 needs and requirements are a subset of the device manufacturer's needs and requirements, then the validation of software for Part 11 also must be a subset of the validation for the device manufacturer's overall needs and requirements. In short, they need not be separate, and they need not be different. The device manufacturer's validation for its ERES needs is likely to include activities beyond what the Part 11 validation would include. To facilitate explaining to an auditor or inspector how the Part 11 validation regulation was fulfilled, the



**Figure 15.1** Device manufacturer and FDA requirements for ERES software.

Part 11 verification tests *could* be separated from the remaining validation for the ERES system, or they could be a variety of verification tests that are traceable to the 36 Part 11 requirements that apply to the manufacturer's intended use of the ERES software. (Remember, the validation is required for intended use. If only the electronic records function is used, and signatures are to be handled outside the software system, then only the 14 requirements related to electronic records would be included in the intended use, and not the 22 requirements related to electronic signatures.)

At the time of this writing, Part 11 is being re-written by the FDA. There is no publicized date for its expected release, nor is there any official word about what the changes are likely to be.

## Nondevice Software Covered by These Regulations

Let's establish some terminology for the rest of Part III. From here forward the automated process software regulated by 21 CFR 820.70(i) and the ERES software regulated by 21 CFR Part 11 will collectively be referred to as "nondevice software". That means software that is not part of a medical device, or is not by itself a medical device, but is used somehow in the design, development, manufacture, or control of quality of a device.

In this part of the text a number of differentiators between nondevice software validation and device software validation will be covered. Those validation differences stem from the differences in tools and methods that are applicable to validation of a given software item used for a given purpose. Why do the tools and methods differ? The range of software types, criticalities, complexities, supplier sources, and levels of control over the software combine to result in a very broad collection of software validation situations, and not all tools apply to all situations.

The GPSV breaks down some of the specific software systems that are within the scope of the 820.70(i) regulation. Specifically mentioned are "device design, testing, component acceptance, manufacturing, labeling, packaging, distribution, complaint handling...." Where did that list come from? Is this list complete? What was meant by "automate any other aspect of the quality system"? What software falls under this regulation, exactly?

It was noted above that the quality system definition is rather vague when applied to software validation. However, one can assume that since the regulations of 21 CFR 820 are the quality system regulations (QSRs), then those items regulated by the QSRs must be the components of the quality system.

Table 15.1 lists the major requirements of a quality system as regulated by the QSRs. The left column contains the section reference number and requirements topic for each requirements topic in the QSRs. The right column contains some ideas for types of software that might automate all or parts of the QSR requirement. This, too, is not meant to be a complete list, but a mind jogger to prompt the device manufacturer to refer to the QSR detail (on the accompanying DVD), and identify what software is used to automate any part of each requirement. That will produce the inventory of potential software items used by the device manufacturer that require validation to comply with the regulation.

**Table 15.1** Some Quality System Components That May Be Automated by Software

<i>QSR Requirement Topics</i>	<i>Possible Software Items That Could Automate</i>
820.25 Personnel training	Training databases, report generation software, OTS applications for training management
820.30 Design controls	Requirements management tools, trace tools, version control systems, CAD/CAM software, software development tools (compilers, syntax checkers, static analysis tools, debuggers, emulators), software controlled test fixtures, automated test tools, defect tracking databases
820.40 Document controls	Documentation control systems
820.50 Purchasing controls	Preferred vendor qualification and status tracking databases, purchasing requirements records, purchase data, lot tracking
820.60 Identification	Product serial number assignment and tracking databases
820.65 Traceability	Product tracking databases, lot tracking databases
820.70 Production and process controls	Production automation systems, monitoring systems, quality control testers, statistical process control software, production process controls, software controlled critical environmental systems, contamination monitoring/control/tracking software, equip maintenance records, inspection databases
820.72 Inspection, measuring, and test equipment	Calibration records and reminders, software automated measurement and test systems
820.75 Process validation	Software automated verification monitoring, spreadsheets, results databases
820.80 Receiving acceptance records	Automated inspection/test, database, tracking, reporting software
820.86 Acceptance status	Database applications and reporting software
820.90 Nonconforming product	Database applications and reporting software
820.100 Corrective and preventive action	Database applications, full OTS solutions for CAPA
820.120 Device labeling	Automated labeling, database, tracking software
820.130 Device packaging	Database applications and reporting software, embedded packaging control software
820.140 Handling	Database applications and reporting software
820.150 Storage	Database applications and reporting software
820.160 Distribution	Database applications and reporting software, full OTS applications
820.170 Installation	Software automated installers, installation qualification automated tests
820.180 Records general requirements	Database applications and reporting software, full OTS applications
820.181 Device master record	Database applications and reporting software, full OTS applications
820.184 Device history record	Database applications and reporting software, full OTS applications
820.186 Quality system record	Database applications and reporting software, full OTS applications
820.198 Complaint files	Database applications and reporting software, full OTS applications
820.200 Servicing	Database applications and reporting software, full OTS applications, service test automation software, remote monitoring, remote update, data logger application software
820.250 Statistical techniques	Spreadsheets, general purpose statistical calculation software, full OTS applications

This table is only sparsely populated with the most obvious possibilities. It is not unusual for device manufacturers to identify thousands of software items that in the scope of the regulations after going through the exercise of inventorying how much nondevice software is used in the organization. Worst of all is that the list is changing daily if not hourly!

## Factors that Determine the Nondevice Software Validation Activities

An inventory only identifies the *number* and location of software items that fall under the regulation. There are other factors that make validation of nondevice software challenging in different ways from device software, and that will determine which validation tools and activities may apply to the validation of a specific software item.

### Level of Control

Size and complexity aside, most of the differences between the validation of device and nondevice software come down to the practicality of what *can be* done. The limitations on what *can be* done depend on the level of control the device manufacturer has on the nondevice software development and testing. Compare the validation of a custom internally developed nondevice software item with the validation of a purchased, shrink-wrapped application ordered from an online source. The device manufacturer has full control over the requirements, design, implementation, and validation of the custom developed software. The shrink-wrapped application may or may not meet the needs of the device manufacturer. The development and validation processes of the shrink-wrapped software supplier are unknown, and usually are off limits to a user of the software.

In the first case, validation of custom, internally developed software has full control of the development and validation activities. The validation of this type of software has all the tools available that are used in the validation of device software. In the second case, the shrink-wrapped software has virtually none of the tools available. What would planning, or reviews look like for shrink-wrapped software? Silly, right? What would a software requirements specification or software design description look like? Could you do unit-level testing? None of these fully apply in the traditional way (if at all) to off-the-shelf (OTS), shrink-wrapped software.

In Table 15.2, some of the traditional validation activities that apply to *device* software are listed. Nondevice software is represented at the top of this table and is categorized by its *type* and *source*. To use the table, determine the type of software to be validated. Find the row representing that type at the top of the table. In that row, select the column that best describes the source of the software. In that column, below the black band are checked the validation activities that may apply to that type of software from that source.

This table is a simplification for the purpose of making the point that fewer traditional validation activities apply to the rightmost columns in the table. The reason is that the level of control decreases for the software for the sources to the right. The table is a simplification. There are exceptions to where many of the check marks

**Table 15.2** Applicability of Traditional Validation Activities to Different Types of Software from Various Sources

Type of Software	Source of Software				
	Internal Custom	External Custom	Open Source	Freeware/ Shareware	Purchased
Out of box			✓	✓	✓
Custom developed	✓	✓			
Custom hybrid	✓	✓	✓	✓	✓
Configurable	✓	✓			✓
Embedded	✓	✓			✓
Programmable	✓	✓	✓	✓	✓
<i>Traditional Validation Activities</i>					
Risk analysis/ management	✓	✓	✓	✓	✓
Needs analysis	✓	✓	✓	✓	✓
Planning	✓				
Requirements	✓	✓			
Requirements review	✓	✓			
Design	✓				
Design review	✓	✓			
Implementation	✓				
Implementation review	✓	✓	✓		
Defect management	✓	✓	✓	✓	✓
Configuration management	✓	✓	✓	✓	✓
System-level verification testing	✓				
Integration-level testing	✓	✓	✓	✓	
Unit level-testing	✓				
Validation testing	✓	✓	✓	✓	✓

were placed in the table. For example, open source software does not have requirements, design, or implementation activities checked. However, if open source software is found with requirement and design documentation, and the device manufacturer decides to review and modify the design and implementation, then obviously more of the traditional activities would apply as the device manufacturer accepts more of the responsibility for the software and takes control of its design and validation (and the software itself starts to look more like it came from an internal source).

As one can see, the applicable validation activities get sparse on the right side of the table. Does that mean that the validation responsibilities of the device manufacturer are less for freeware and purchased software because fewer traditional validation activities apply? The answer has to be a resounding “no.” If that were true, then purchasing software or relying on freeware or shareware would represent a validation shortcut or loophole. The result would be the unintended consequence of creating an incentive to use what might be the least reliable and least maintainable software simply to avoid validation!

If traditional validation activities don't apply, then what is a device manufacturer to do to validate software over which there is little control? When traditional activities don't apply, nontraditional activities must be considered. That is the topic of much of the rest of Part III.

Before getting too far away from Table 15.2, a few words are appropriate to explain what is meant by software type and source as used in the table.

### Type of Software

These types partially align with the GAMP [2] categories of software. There is no regulatory or other requirement to label a software item with a *type*. Any given type of software is usually only available from a subset of sources from which one can acquire the software. As noted above, type and source together determine the level of control one has over the software development and validation activities. Examples of types of nondevice software include:

- *Acquired for Use*: Also known as off-the-shelf (OTS), commercial off-the-shelf (COTS) software, out-of-box, or shrink-wrapped software, these are applications acquired for a specific purpose (training database software, defect management systems, etc.) from an external source and used out-of-the-box, unmodified. This type of software may have substantial user bases, but suppliers may be slow or reluctant to respond to perceived problems or suggested improvements.
- *Custom developed*: Developed by the device manufacturer or for the device manufacturer for a specific purpose (automated custom test fixtures, custom complaint management systems, one-of-a-kind production automation equipment with embedded software, etc.).
- *Customized/hybrid-partially OTS*: Large installations may have OTS software modified for their specific needs by contracting the OTS supplier to add features, modify existing functionality, or defeature the software.
- *Embedded*: Software or firmware that is embedded in an instrument, test device, machine tool, or production machine, or any other device that could impact the quality of the medical device being produced.
- *Configurable*: Certain complex OTS software is designed in layers, and with configurable feature sets, drivers, and so forth. These are different from standard acquired for use software items because a large number of configurations may be possible, many of which may not have been anticipated or tested by the supplier of the software.
- *Programmable*: Software that is OTS, but has no identifiable intended use until a "program" with a specific purpose is created by the user. Examples of this type of software include spreadsheets, statistical analysis software, mathematical analysis software, instrument control software, macro controllable software, and so forth.

### Source of the Software

The selected *source* (i.e., the developer) of the software will determine how much control the device manufacturer has over the specification, development, and vali-

dation of the software. Understanding the impact on level of control is important for two reasons:

1. *After-the-fact (retrospective) validation.* Unfortunately, it happens. Device manufacturers entrust their data or procedures to software, *then* think about what they should do to validate it. Level of control determines what tools are available to them at that point.
2. *Prospective validation.* Device manufacturers may (hopefully) be prospective enough in their thinking that *before* they acquire or develop software they consider the risks of trusting the data or process to software. The identified risks should be balanced with the level of control that will be possible for software being considered from different sources. The level of control will determine the validation tools that are possible given the source. One can, even before the software is acquired, determine if the confidence in the source of the software after validation would be sufficient to reduce the risk to an acceptable level. If not, perhaps another source for the software should be considered.

Consider the example of software that will process data to identify report or complaint trends that indicate serious defects in the medical device in the field. It is a critical process. Failure of the software could lead to delays in identifying device defects. Suppose the software is available as freeware from a hospital technician somewhere in Eastern Europe. A device manufacturer is considering using the software, or developing a custom application for the same purpose. The device manufacturer must decide whether the available validation tools for the freeware would be adequate to provide enough confidence in the software for this critical function. Otherwise they may opt to custom develop the software.

Some of the common sources of software include:

- *Internal:* Developed internally by employees or closely controlled contractors.
- *Outsourced:* Contractually developed by a development house offsite, typically with less control than internally developed software.
- *Purchased:* OTS applications with no control over requirements or development and test methodologies, but some contractual obligation for being fit for use.
- *Open source, shareware, freeware, and SOUP:* “As is” applications with no control over requirements, development, or test methodologies and no contractual obligations for being fit for use. Software of Unknown Provenance (SOUP, a 62304 Standard term) comes from unknown sources.
- *Combination:* A software system may be comprised of elements from different sources. For example, an outsourced application, may be developed under contract, but could include some open source graphics libraries, and a purchased OTS database engine.

### Other Factors That Influence Validation

The level of control one has over the nondevice software will determine what “traditional” validation activities are *possible*. Traditional activities are those activities

one traditionally considers when *developing* the software. In later chapters, validation activities other than the traditional activities will be considered to round out the “validation toolbox” for nondevice software. Once the *possible* validation tools are understood, one must decide how many of those tools will be applied, and in the case of some tools, to what level of rigor they will be applied.

That is where other factors enter into the validation planning activity. Some of these factors should be familiar from the techniques used for device software. Some are more specific to non-device software. Let us look at some of these other factors.

## Risk

Risk should always be at the top of the list. Risk analysis for nondevice software is somewhat different from risk analysis for device software. The control measures that one might consider for risk management also are different. This subject will be given a full treatment in an upcoming chapter, so the discussion here will be held to a minimum. Even though the specifics may be different, the general process will be similar to that of device software. Higher risk applications will choose to apply more validation tools with more rigor than lower risk applications.

## Size and Complexity

Nondevice software validation can be challenging because software items as small as a simple spreadsheet or a macro to massive global IT solutions all fall under the same regulation. It really defies a simplifying the validation to a check list of activities that must be done.

For software that is not custom developed, the device manufacturer may not even be aware of how big the software is that is being used, or how complex the underlying code is that implements the features. The device manufacturer *is* aware of how complex the process is that is being automated or partially automated by the software. Validation activities should be planned to protect the device manufacturer from failure of software in the context of the process being automated. In other words, the *amount* of validation of the full process that is automated (or partially automated) by software will naturally be related to the size and complexity of the *process*.

## Intended Use

It is important to understand what functionality in nondevice software needs to be validated, and, just as importantly, what functionality does not need to be validated. Thinking of intended use of nondevice software always should be done in the context of the process that the software is intended to automate or partially automate. Often, this will lead to investing some significant time into developing an understanding of the process itself first, then understanding the role of the software in that process; that is, its intended use.

There is much more to be said about intended use, but a more detailed discussion of intended use is deferred to a later chapter on intended use and requirements. However, in summary, a clear understanding of intended use will be necessary to adequately identify the risk of using the software. Further, the complexity of the process being automated and the role of the software in that process is defined by its intended use.

## Confidence in the Source of the Software

In the Chapter 8 coverage of risk for device software, several “soft factors” were mentioned that affected the likelihood of software error. Two of those soft factors had to do with the skill and experience of those who developed the software. Certain things can be done to reduce that probability of error due to lack of skill or experience, thus partially controlling the risk. Also recall that the confidence that validation helps build comes from activities that prevent defects (through design controls), and that find and eliminate those defects that do find their way into the software.

That all applies neatly for nondevice software that is custom developed by the device manufacturer, but what happens when that software is purchased or downloaded from a shareware web site? The preventive confidence building activities are no longer in play. The software was already completed when it was first found or purchased. Does that mean that one is left only with finding defects; that is, through testing, or field experience? What happens if that testing does find a defect? What recourse is there then?

The confidence one has in the source of purchased or free software will have to substitute for the confidence that one would have in one’s own development team. There are ways to establish just what that confidence level is, but that is only half the question. One must also be prepared with decisions or compensating actions if that confidence level is not acceptable. Those decisions may include looking for other sources, and the actions may include additional testing, source code escrows or other external risk control measures in case the software is flawed.

## Intended Users

The intended *users* of the software should be taken into account for validation of both device and nondevice software. However, there is a difference. With device software, the intended users tend to be classes of users: patients, clinicians, and so forth. With some nondevice software the users may be individuals, and could even be a single individual.

Why does this matter? The difference has to do more with the number of users. Not all nondevice software is intended for widespread use. Some software, such as spreadsheets and macros, may be used only once by the creator of the software (single use—single user). Should the validation of a single-use spreadsheet used to calculate a manufacturing process variable be the same as the validation of a spreadsheet performing the same calculation but used by a large number of users multiple times?

One could argue that the same calculation is done in both, so the validation should be the same. Another way of looking at this question could conclude that the severity of a failure is the same, but the probability of an error is higher if multiple users are involved.

Why would the number of users affect the probability of error? A user unfamiliar with the software might accidentally enter erroneous inputs, or accidentally change a formula, or may not recognize when the answers are grossly incorrect. As the user base increases in size, so does the diversity of experience and skill. The likelihood of an error due to erroneous input increases with the number of users.

But wait, we are now talking about the likelihood of user error. What does that have to do with software validation? The software design may have been ambiguous in directing the user, or just wasn't designed to detect and survive a user error. Is it a software error if user error is foreseeable and likely, and the software confuses or misleads the user, or does nothing to attempt to prevent the user error, or to detect and correct the user error? Perhaps it is not a *programming error* but it certainly is a *design oversight* that validation should identify and correct.

Let's return to the topic of how the number of users of a software item should impact the validation of that software. If one accepts that anticipating user error and designing control measures is part of validation, then the data of Table 15.3 shows an example of how the number of users can impact the likelihood of encountering an error.

The values in this table are based on the average probability  $P_u$  of any given user making an erroneous entry. Assuming each user uses the software only one time, the probability of at least one user in the population of users making an erroneous entry is  $1 - P_0$ , where  $P_0$  is the probability that nobody makes an error. For any given usage of the software a user will make an error with probability  $P_u$ , or will not err with probability  $1 - P_u$ . In a population of  $n$  users, the probability of all users not making an error is  $(1 - P_u)^n$ , which is  $P_0$ . Therefore, the probability that at least one user will make an error is:

$$(1 - P_0) = 1 - (1 - P_u)^n$$

The probability,  $P_u$ , is represented in the left column of the table in its reciprocal “odds” form. That is a probability of 0.1 is equivalent to odds of one in 10, and is shown in the table as a 10. So, with odds of one in ten of an individual user making

**Table 15.3** How Number of Users Affects Probability of Failure

Odds of Failure (1 in “x”)	Number of Users					
	1	10	100	1,000	10,000	100,000
2	50.000%	99.902%	100.000%	100.000%	100.000%	100.000%
10	10.000%	65.132%	99.997%	100.000%	100.000%	100.000%
100	1.000%	9.562%	63.397%	99.996%	100.000%	100.000%
1,000	0.100%	0.996%	9.521%	63.230%	99.995%	100.000%
10,000	0.010%	0.100%	0.995%	9.517%	63.214%	99.995%
100,000	0.001%	0.010%	0.100%	0.995%	9.516%	63.212%
1,000,000	0.000%	0.001%	0.010%	0.100%	0.995%	9.516%

an error, there is a 100% probability (rounded to three decimal places) of the user error showing up if there are 1,000 or more users.

Clearly, the number of users increases the probability of failure. It is interesting to consider a single user who also is the creator of the software. The multiuser scenario above used an average probability of user error of  $P_u$ . The user/creator of the software is *not* as likely to be confused by what the software is expecting for inputs, will know what units to use, and is more likely to recognize a crazy result if he does err. Consequently, one could make a good argument that the probability of user error for the single user/creator is much lower than  $P_u$ .

For the sake of argument, let's assume that the creator is 100 times less likely to err than the average user. Let's compare the results for a population of 100 users to that of the single user/creator. Assume for this example an average probability ( $P_u$ ) of error of the inexperienced user of one in 1,000, and the odds that the creator of the software errs is 100 times less or one in 100,000. Again, referring to Table 15.3, the probability of at least one of the multiple users making the error is 9.521%, but the probability of the single user/creator making an error is 0.001%!

It seems practical that there should be some difference in the validation of the multiuser spreadsheet, but what?

Is additional testing the answer? More testing would not hurt, but at least some of the problems anticipated were user errors. It is at this point that many would say, "Well, user errors are much more likely than software errors so we are wasting our time validating the software." Maybe that would be true if validating were equivalent to testing only. However, nondevice software validation includes many activities that are not test activities. The same was true for device software validation.

So, what validation activities would save us from user error? Risk management is a validation activity—part of the validation toolbox. Part of the risk management activity could identify erroneous user inputs by inexperienced users as a potential hazard (depending on how the result of the calculation is used). The risk control measure might be to do some range checking of the inputs to alert the user of erroneous inputs. Labeling could be added to clarify the expected units of the input, or the software could deal with multiple units of measure and ask the user to specify which unit is used. To control the risk resulting from a mistakenly modified formula, the formula could be locked in the spreadsheet so unauthorized users could not change the formula.

Wait! Those are design changes, not validation, you say. True, but the requirements for the design change came from a validation activity—hazard analysis and risk management. Recall earlier the text noted the overlap and fuzzy distinction between validation and good engineering process. This is a good example. Did the solution come from design and development, or from validation activities? It is hard to distinguish because the two are so intertwined, and it really doesn't matter. The next iteration of validation activities may test those control measures to be sure they are effective. Confidence in the software is enhanced. The validation made the software safer, and the validation time was not wasted.

If the risk analysis leads one to the conclusion that those control measures are necessary, wouldn't they apply whether the spreadsheet is used by multiple users or just the spreadsheet creator himself? It depends on other circumstances one of

which may be the question of how many times the software will be used and over what period of time. This again is related to risk. The risk of erroneous entry or formula modification increases over time. Let's say the spreadsheet is used only twice a year. In 6 months, is there a risk the user/creator might forget the units of data entry, or may not recognize an erroneous result? If so, then the multiple user risk control measures may be appropriate even for the single user.

Now consider that the single creator/user just created the spreadsheet, wants to get the result, and then will never use the spreadsheet again. Would the same erroneous input risks apply? Probably not. Would the same multi-user risk control measures be appropriate? They would probably be overkill.

Would it hurt to impose those multiple-user controls on the single-user scenario? It wouldn't hurt the software. It would take more time for the creator to get his or her quick answer from the spreadsheet. So it would be overly conservative, but that's OK, right?

Yes, that's OK for this spreadsheet. However, there may be a bigger issue at stake. If overly burdensome validation processes are imposed on an organization, the unfortunate response often is to "go underground" with software. That is, users may hide software, or not admit it is being used to *avoid* validation. That clearly does not achieve the confidence building in the software that validation was meant to achieve. Hiding software doesn't make one more confident that it is working correctly and reliably.

Overly burdensome validation requirements may lead to another type of unfortunate response which is that users may refuse to use software tools in their day to day work opting instead to use pen and paper techniques which are not only less efficient, but also more prone to error than the software itself.

A device manufacturer's quality system has to strike a balance that allows those validating the software to select the tools that apply, commensurate with the risk of using the software. If validation has no perceived value in the eyes of those doing the validation, they won't be doing it for long, and they probably won't be doing it well while they are.

How do these various attributes make nondevice software validation different from device software validation? Device software validation also varies with size and complexity, but not over the wide range that nondevice software does. Nondevice software runs from simple single page spreadsheets to multimillion line IT applications. The challenge for device manufacturers is to come up with quality system processes that adequately deal with the vast diversity of software items that are subject to these regulations.

## Industry Guidance

Often when there is a regulatory requirement but little documented guidance on how to comply with the requirement, an industry-initiated guideline is proposed. For the validation of nondevice software, two industry organizations have sponsored such reports or guidelines. The Association for the Advancement of Medical Instrumentation (AAMI, [www.AAMI.org](http://www.AAMI.org)) released in 2007 a technical information report (TIR) on this subject specific to the medical device industry. A second organi-

zation, the International Society for Pharmaceutical Engineering (ISPE, [www.ISPE.org](http://www.ISPE.org)) has released several versions of their *Good Automated Manufacturing Practice* guide (more commonly referred to as GAMP).

### **AAMI TIR36:2007: Validation of Software for Regulated Processes**

Validation for each of the types and sources of software require different methods or “tools.” The problem of validating nondevice software is that it is so broadly defined that no single method or tool will work for all circumstances. One of the resources that will be mentioned often in this part of the text is an AAMI Technical Information Report (TIR36) [1] that covers the topic of the validation of nondevice software. TIR36 takes a toolbox approach to validation. That is, only the tools that apply to the specifics of the software and its intended use are applied to the validation at hand. That does seem obvious, doesn’t it? It is surprising how long it took the workgroup to come to that realization.

As one of the contributors to TIR36, I can report that the “obvious” toolbox approach took the work group of 14 “experts” about 2 or 3 years to recognize and document. There are many reasons it took so long (for one, the meetings were only a few times a year), but one reason is that the experts each had a different view of what the right approach was for the validation of certain types or sources of nondevice software and tried to apply that approach to all types of software.

When the TIR36 workgroup was first convened, it was common in the industry to create reams (literally) of test documentation for common applications. Practically the only validation activity employed was testing. Little of the testing added significantly to the confidence in using the software. It was becoming recognized that the validations that were being done were expensive, time-consuming, and were not adding any value to the confidence building. The workgroup’s mission was to give the industry some guidance in improving the way nondevice validation was undertaken.

After about 3 years of debate and authoring the report, the workgroup decided to add examples of validation plans to show how the toolbox method could be applied. The group initially expected that all the examples, which were assigned to various workgroup members, would have a consistent look as the common method was applied. However, even after years of working together on the methods in the report, the examples, when collected for review looked very different from each other. Initially, there was some discussion about revising them to look consistent, but, in the end it was recognized that each approach did accomplish its validation, and that perhaps a better message to deliver in the report was that there is more than one way to solve the problem—even if you are using a common methodology!

### **GAMP 5: Good Automated Manufacturing Practice**

GAMP comes from the pharmaceutical industry to address its need to validate production automation software in that industry. The FDA regulations for devices, as seen above, are broader in scope than just production software, so this text will refer primarily to the AAMI TIR since it is specific to the medical device industry. The GAMP guideline is, nonetheless, an excellent resource on this topic. GAMP 4

was studied carefully by the AAMI TIR36 workgroup, and elements of GAMP 4 were adapted for use in TIR36. The latest version, GAMP 5, focuses more on risk-based validation activities than its predecessor.

## Who Should Be Validating Nondevice Software?

Let's take on this last question before moving on to detailing some of the activities that are specific to nondevice software validation. Who should be doing all this validation?

Earlier in this chapter some ideas were presented for what software applications might be included in the scope of the two regulations that require nondevice software validation. That was a long list, and, admittedly, it was not all-encompassing. Companies are known to try to inventory the total number of software items that fall under these regulations. The number of software items is in the thousands.

Other companies have done their own inventories and have come up with numbers orders of magnitude smaller. Which inventory count is “legally” correct will ultimately be decided by the regulatory agencies. However, it is doubtful that a company of thousands of employees has only a few dozen software items within the scope of the regulation.

That is an important regulatory issue, but going back an innate distrust of software, does it matter if the software is within the scope of regulation? Shouldn't a company care if it is using software that could injure a patient—or thousands of patients? Shouldn't a company care if there is software in use, the failure of which could cost the company millions of dollars in rework, recall, not to mention the loss of reputation? Shouldn't an individual care if the calculations made for his or her own work are correct? Does everyone really trust the software that much that they think validation is a waste of time? Worse, is everyone just afraid of what they will find if they start looking for problems?

One of the problems in the industry today is that except for very few companies, the task of validating nondevice software is nobody's *responsibility*. It usually just isn't done unless it is a big obvious software system that is likely to attract regulatory attention. Sometimes somebody within the company with regulatory responsibility begins to worry about the regulatory risk of not validating and not even having the appearance of validating nondevice software. That is a sorry state for the industry to be in. Validation is driven by regulation, not concern for safety, or concern for business risk. Shame.

*If* a company does decide to take this seriously, then who should be responsible for it? Large-scale internally developed applications probably are, or should be validated by an internal validation team, and that validation will look much like a device validation. That's the easy answer. What about those hundreds, or thousands, or hundreds of thousands of software items like spreadsheets? What about that great new software application someone just found out about that is perfect for a specific need at work? Who validates those? There probably aren't enough software savvy people in a company to handle all the validations necessary if every little validation falls to a specialized validation team. If not the software developers or software quality assurance team, then who?

One answer has to be that the user himself or herself will have to take on more of the responsibility for validating software that he or she uses. Who is a better expert on the intended use than the user? Who will notice if things don't work or don't look right better than the user? Does that mean that everyone will have to go through a 6-week training program on software validation? The user is the best person to validate most small to medium scale software items. Excessive training sessions clearly won't be reasonable, so there needs to be a validation model for software users—a do it yourself (DIY) software validation for software that is acquired through purchase or from no cost sources.

That leaves us with the externally developed custom applications and those software items with a level of control somewhere between totally customized internally developed software and the DIY software validation items. As you might expect, those validations will look like an enhanced DIY validation, or a defeatured version of the full validation done for internally designed custom software. For each of those, the critical thinking must be applied to choose from the validation toolbox those tools that best fit the software type and source, as well as the risk, complexity, and user profile.

The following chapters of Part III will cover a proposed model for DIY software validation, and the application of critical thinking for software items requiring validation, for which there is not a template model.

There is one other party that should be considered as a responsible party for validation of nondevice software. That is the supplier of the software. The supplier's validation will not fully replace the device manufacturer's validation since only the device manufacturer will know the intended use of the software. However when the supplier performs validation activities that are documented and were done at an acceptable level of rigor, then at least they can substitute for some of the activities that would have to be done by the device manufacturer.

## Reference

- [1] AAMI TIR36: 2007, *Validation of Software for Regulated Processes*.
- [2] *Good Automated Manufacturing Practices (GAMP) 5: A Risk-Based Approach to Compliant GxP Computerized Systems*, ISPE 2008.



# Planning Validation for Nondevice Software

## Introduction

In the first chapter it was suggested that if one feels that time is being wasted in validating software, then it probably is a waste of time. Not that validation is a waste of time, but that the particular validation activities that lead to that feeling probably are a waste. One of the reasons time is wasted in validation is that the activities are not conducted in the order in which they add value (e.g., why, other than for regulatory reasons, bother writing requirements *after* the software is written?). Maybe there is a misunderstanding of what the activity is meant to be, or a lack of understanding of why it is being undertaken. That is one reason there has been an effort in this text to explain *why* the activities are suggested.

Another situation that can lead to that sense of wasting time is when an organization tries to come up with a one-size-fits-all approach to validation. Nondevice software is particularly resistant to this approach, which has also been referred to a number of times in this text as the “checklist approach.” It is just impossible to come up with a single process or procedure that works equally well for all software items in the broad range of software types, sizes, and complexities—unless that process is extremely complex and filled with exceptions. Even then, it would be a compromise solution at best (and, as experience has shown, highly unlikely ever to be read).

Does that mean we should just give up on validation then? That’s probably not a good alternative, so the approach to validation needs to change to one in which one thinks critically about the options available for validation; that is, the validation toolbox. Conveying critical thinking through long procedures or checklists that don’t always fit the problem just doesn’t work.

I ran across the following quotation in another book recently, and it is especially appropriate to this topic of overly burdensome processes that do not deliver the desired results.

Simple, clear purpose and principles give rise to complex and intelligent behavior.  
Complex rules and regulations give rise to simple and stupid behavior.

—Dee Hock, founder and CEO of VISA credit card association

There is a level of comfort in the checklist approach. Checklists are important reminders for us to check for the most common problems, or to perform the most basic subset of activities. It is when the checklist stops being just a reminder of the

basics and substitutes for critical thinking that it becomes problematic. Checklists are also comforting because it is clear when a job has been completed successfully—every box has a check in it, regardless of whether the software is meeting needs.

How should one define the successful completion of the validation of a nondevice software item? In the Foreword to TIR36 [1], this suggestion is found:

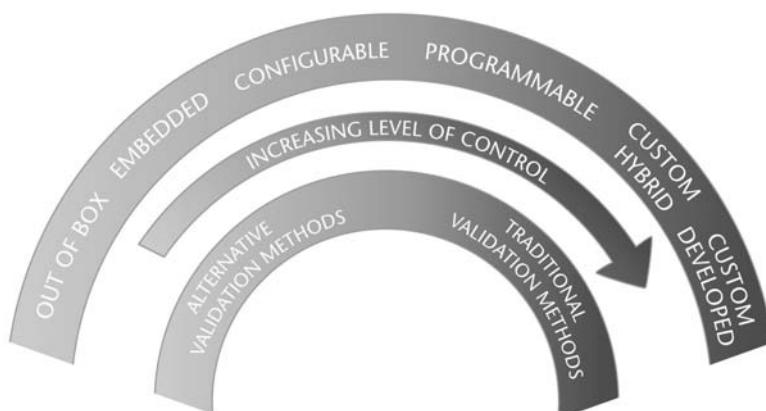
For the software validation efforts to be viewed as highly successful, the following statements should be true:

- The automated process or associated software functions as intended, without compromising device safety, device quality, or the integrity of the quality system.
- The people performing the necessary activities believe that the efforts are meaningful and worthwhile (i.e., least burdensome or most valuable activities).
- The manufacturer is in a state of compliance.
- Auditors and inspectors view the activities applied and the resulting records as acceptable evidence of compliance.

## Choosing Validation Activities

Figure 16.1 shows that validation planning and the choice of appropriate validation activities is a multiparameter decision. There is a very broad range of software types. There are hybrids of those types that are possibly from multiple sources for the same software item being validated. This dictates which validation tools it will be *possible* to use. The ranges of size, complexity, intended use, risk, and confidence in the source of the software moderate how many tools are to be applied, and how rigorously they will be applied.

Let's examine the two ends of the spectrum to start the discussion on choosing validation activities. The right side of the spectrum represents the custom internally developed software. The choice of validation activities is less restricted in this case because there is full control of the specification, design, and development of the software. In fact, the validation of the software could look almost exactly like the



**Figure 16.1** The spectrum of nondevice software validation.

validation of device software. There are a few differences in some of the activities between device and nondevice software and those differences will be discussed shortly. However, all of the confidence-building validation activities discussed so far in this text are available for application.

The validation moderators mentioned above (size, complexity, intended use, risk, and confidence in the source) can be referenced as part of the critical thinking that chooses from among the full set of validation activities. Why wouldn't one always choose all of the activities if they all are valuable and all increase our confidence in the software? At some point, validation commensurate with complexity and risk, return on investment, or the law of diminishing returns enters into the critical thinking. That is, sometimes additional validation activities are out of proportion to the software being written, or are just not worth the cost, or are unlikely to further build confidence.

Consider a half page of Visual Basic software written by a production engineer to collect production process data for a regulated process. The prospective author may decide to write some tests to be sure the software works. Hopefully the author would consider the other do-it-yourself (DIY) activities discussed below. Would a 3-page development plan be likely to add (return) any value for the investment in time? It probably would not. Would a design review or code inspection make sense? It might, depending on the criticality and/or complexity of the software, or the experience level of the engineer. If a simple review concludes that it is low-risk software, very simple, and its failure would be clearly detectable and recoverable, then anything more than a simple review may not add much value. Would 150 pages of test procedures be likely to find any problems that wouldn't be found in 10 pages, or 5 pages? The point of this questioning is to point out that although all the validation activities are available for internally developed custom software items, it doesn't always make sense to apply them all, or apply them to a level that is disproportionate to the size, complexity, or risk of use of the software.

Questioning similar to the example of the preceding paragraph is what those charged with validating nondevice software should ask themselves in selecting validation activities; that is, validation planning. This kind of validation planning leverages the critical thinking that TIR36 teaches. There is a subtle point to make. Critical thinking can become a loophole when the thinking's primary objective is to simply reduce the validation effort. Instead, the critical thinking should be more positive. Critical thinking should be focused on adding available activities until confidence in the software is achieved, not on removing activities until confidence is shaken.

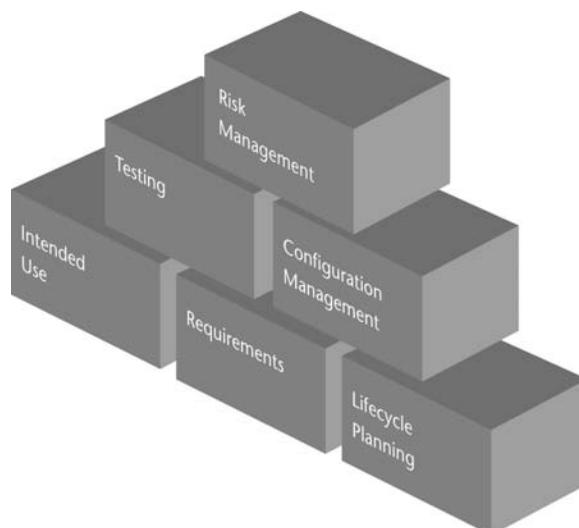
## Do-It-Yourself Validation or Validation for Nonsoftware Engineers

Now, think about the other end of the nondevice software spectrum, the out-of-the-box acquired for use, unmodified software. In most device manufacturer's organizations there probably is not any one person or department responsible for validating this kind of software. Users purchase the software to use it to solve a problem or to automate a process. They probably are not thinking about validation. How does this software get validated?

The user himself or herself oftentimes must step up to handle the validation in these cases. User-validation may be the only sustainable nondevice software validation model that will work for off-the-shelf software and small self-created software items like spreadsheets. Sustainability is an issue because there may not be enough software engineers or validation engineers in the world to handle the validation workflow otherwise. How is someone with no software background supposed to do this?

For these types of user-validation there are six fundamental ingredients of a validation effort. They don't have to be complicated with multipage documents. In fact, all six items may easily fit in one document. Those six basic building blocks, shown in Figure 16.2, are:

1. Intended use: The problem the software solves, or the part of a process it automates.
2. Requirements: The specific requirements of the software to successfully fulfill its intended use.
3. Risk: An analysis of how failure of the software could result in failure of a medical device, or failure to comply with any regulatory requirements and a plan for reducing the risk to an acceptable level.
4. Life cycle planning: Identification of the phases of a life cycle for the software that models the life of the software from acquisition, to retirement, and a plan for what activities will keep the software in a state of validation.
5. Configuration management plan: A plan for how problems with the software will be reported, how changes to the software will be managed, and how unauthorized changes to the software will be detected.
6. Testing: Verification that the requirements have been fulfilled and that the software satisfactorily meets the needs of the user. This may be achieved by customized testing, reference to 3rd party testing, reference to supplier testing, user base experience, or a combination of some or all of these possibilities.



**Figure 16.2** Basic building blocks of DIY nondevice software validation.

These basic building blocks will be discussed in some depth in the upcoming chapters, so further treatment will be deferred to those chapters. For now, just recognize that the basic building blocks should not represent “a waste of time” in new overhead. These building blocks address the questions that the typical responsible user asks himself or herself already, and document that thinking in a form that is convincing to another observer. These questions include:

- What is this software automating? Why do I need it? (intended use);
- What functions and features do I want the software to have? (requirements);
- What if the software doesn’t work? What could happen? What should I do? (risk management);
- What should I do to be sure the software works and continues to work? (Life cycle planning);
- How do I know whether to install software updates? How will I know if they are installed automatically or by someone else? Will it matter? What if I find errors? (configuration management);
- How do I know the software is working? (testing).

If one has asked these questions and has resolved them satisfactorily, then a good level of validation has been achieved. All that is left is to document that process. Why? First, it should be documented for objective evidence for auditors and regulators that you did indeed validate the software. Second, documentation provides a record of what intended uses were considered in the validation (especially for risk management) so that if new uses evolve or new users materialize, one can reassess the state of validation for that new use or user. Third, there is something about committing a thought process to writing that helps identify the gaps and faulty logic in that thinking. If nothing else, it allows for other parties to review the logic to help find problems and suggest solutions.

## The Nondevice Software Validation Spectrum

The software spectrum of Figure 16.1 implies a spectrum of validation methods and styles that apply to the various types and sources of software. The custom, internally developed software items have full control of the software and thus can consider the full range of validation life cycles and validation activities used for device software that were described in Part II of this text. However, even for custom-developed software that is simple, low-risk software, one appropriately may opt for a subset of those activities or substitute activities that are more appropriate for the size, complexity, and risk of use of the software.

At the other end of the spectrum the out-of-box software and embedded instrument software offer very few or no controls during the development life cycle. The medical device manufacturer is not developing them or even specifying their development. Yet they may be very large and complex with a high risk of use. Further, despite the high risk and complex nature of some of this software, the responsibility for its validation may be assigned to users of the software who are not validation or software experts. This is the DIY end of the spectrum.

In between the full device software validation models used for medical devices and the DIY validation model are a large variety of other software items that do not neatly fit into either category. Configurable software is configured by the user, but often of off-the-shelf components. Programmable software is a custom software component that runs within an off-the-shelf software component (for example, spreadsheets/custom and Excel/OTS.) Some validation attention must be given to both components. Custom hybrid software also is software that is made of various components, some of which may be custom, and others of which may be off-the-shelf, or any other type of software. Validation of all of these midspectrum software types need careful analysis to understand the risk associated with each component, and the risk management and validation activities that are possible (based on level of control) for each component to craft a validation plan that is appropriate. As mentioned before, this takes critical thinking and defies a simple checklist approach. Device manufacturers may try to define the scalability of validation across the spectrum of software types in corporate procedures. In fact, it may well be necessary for larger corporations to provide some guidance in the form of recommendations from the toolbox menu for specific types, sources, complexities, numbers of users. This allows flexibility for the experienced and guidance for the inexperienced.

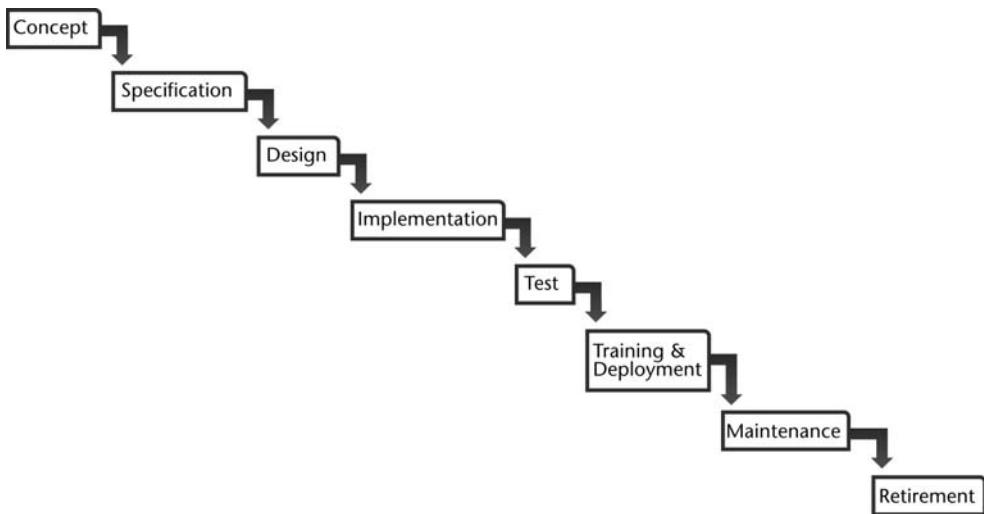
## Life Cycle Planning of Validation

One of the building blocks referred to above is life cycle planning. One of the major advantages of using a life cycle approach to either development or validation is that it organizes one's thinking about the activities that are applicable at each phase of the life cycle. It is the first that will be described in some detail, because the other validation activities will be organized around the software's life cycle model.

Nondevice software can be described by a number of different life cycle models. Unlike device software that is developed by the device manufacturer, nondevice software's life cycle model choice is more often dictated by the circumstances of its acquisition and level of control than over personal preference. The variety comes from the broad range of types of software and sources of that software.

In the following, a few examples of nondevice software life cycle models are presented. There are many more examples one could imagine. The broad range of nondevice software is best modeled with a broad range of life cycle models. In fact, it is problematic for companies to attempt to squeeze all nondevice software validation into a single model. It just doesn't work. Forcing it to fit will result in time wasted in figuring out how to make it fit, and likely will result in at least some effort and documentation that has little value because of working around a life cycle model that is too restrictive.

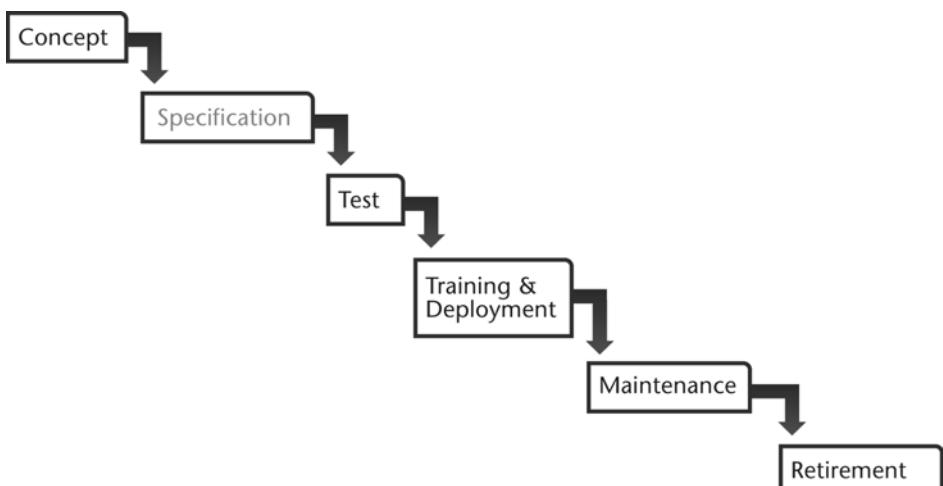
The traditional waterfall model of Figure 16.3 is a good fit for internally developed custom software. At least, it is as good a fit as it is for device software. The objections to traditional waterfall life cycles for device software will not be repeated here, but most would apply equally to the development and validation of nondevice software. The important thing to notice is that some of the phases, and hence the



**Figure 16.3** Traditional waterfall life cycle for internally developed nondevice software.

activities that would take place in those phases, only make sense when one has full control of the specification, design, and implementation of the software.

Suppose the very same nondevice software is to be developed, but the company elects to have the software developed by an outside software development firm. The controls over design and implementation may no longer be an option. The life cycle for the externally developed software might look like that shown in Figure 16.4 (again ignoring all the usual objections to traditional waterfall life cycles). Suppose the device company only gave the outsourced development company a rough concept of the software, and not detailed requirements specifications. They then lost the opportunity to define their own requirements for the software. The validation activ-



**Figure 16.4** Traditional waterfall: externally developed.

ties (review of requirements) of the specification phase might be lost as indicated by the faded color of the specification phase.

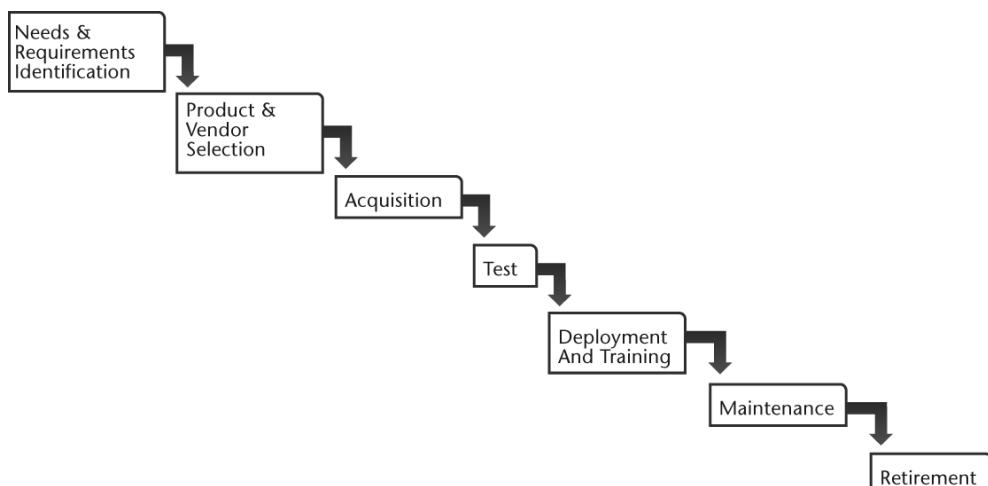
Now suppose the software is available off-the-shelf from a number of vendors, and the company decides to purchase the software off-the-shelf and plans to use it as it comes out of the box. The life cycle of Figure 16.5 may be more appropriate. Still a waterfall-type structure, the life cycle has very different names for the early phases. That suggests that the validation activities within those phases may be very different. In particular, notice that the design and implementation phases have been replaced with product and vendor selection and acquisition phases. The activities related to building confidence in the software will be very different from those activities that are possible when designing one's own software.

Now consider the example of some simple single-user spreadsheets that the device manufacturer needs to create. This software is really two components, the off-the-shelf spreadsheet software (e.g., Excel or other commercial or open source alternatives), and the individual spreadsheets themselves. Consequently the life cycle of Figure 16.6 shows one life cycle track for the base spreadsheet software, and several repetitions of another track for each spreadsheet created.

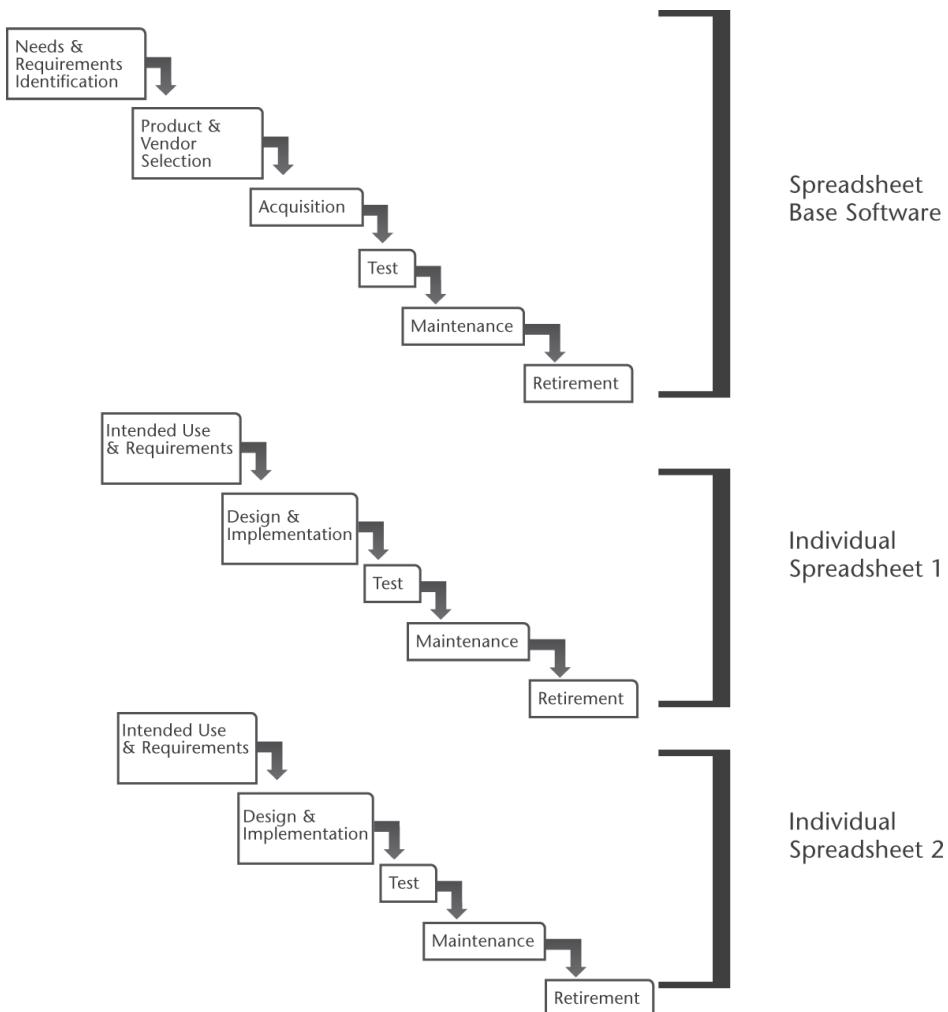
Hopefully, it is starting to become clear why different software items might have different life cycle models, and why the validations may use entirely different tools. This is a good first step in the critical thinking needed for nondevice software validation. However, it is only recognition of the life cycle of the software in the organization. Before validation planning can be started, a new box of tools is needed for validating non-device software.

## The Nondevice Software Validation Toolbox

Why is a different set of validation activities (i.e., tools) needed for nondevice software? There are two reasons:



**Figure 16.5** Life cycle model for purchased out-of-the-box nondevice software.



**Figure 16.6** Life cycle model for acquired but programmable software.

1. To replace validation activities lost with loss of control of the software. Except for internally developed custom nondevice software, some level of control of the software has been lost. Some of those controls were validation activities. With reduced validation activities available, confidence in the software may be questionable or unacceptable. Additional tools are needed to reestablish confidence that the software is fit for use.
2. To take advantage of validation methods that may not be available for device software. Nondevice software is typically used within the confines and under the control of the device manufacturer. Therefore, the *use* of the software may actually be under *more* control than device software whose use is controlled by the device user. Consideration of the role of the software in the process it is automating or partially automating may lead to verification or validation activities that can lead to acceptable levels of confidence.

What is in the validation toolbox? For starters, all the validation activities that were mentioned under the topic of device software validation are available—if they are applicable to the nondevice software in question. A number of other tools are mentioned below. They are important and valuable validation tools, but they are by no means the only tools available. Creativity is a good companion to critical thinking when it comes to the validation of nondevice software. One should always think critically about the risks imposed by using the software, and think creatively about how to build confidence that those risks have not or will not materialize. The following list and the suggestions that can be found in TIR36 should be used if they fit the situation, but they should also be used to stimulate the creative thinking about other methods and activities available.

### **Product Selection**

When software is acquired (purchased, open source, shareware, etc.), there is an opportunity often to select among multiple available options. Not only is it an opportunity to choose the product that best meets the needs, it is an opportunity to learn about other users' experience with the software. Internet-based user groups, complaint blogs, and review articles offer easily accessible information that anyone can access to build (or destroy) confidence in a product. Don't forget the "objective evidence" if the selection process is to be used as one of the validation activities.

### **Supplier Selection**

If little is known about the software, then one can research the vendor of the software. Internet-based research can determine if there are many complaints about the supplier of the software. Just knowing about the supplier can help form an opinion about the risk of using the software. For example, if the supplier is running a cottage business and working on the software as a moonlighting job, one might ask what kind of support and responsiveness could be expected should problems arise.

Even if the supplier is a commercially viable enterprise, that doesn't necessarily mean that the software and support will be a positive experience. When one's only or best option is to rely on an external source for software, the GPSV suggests:

Where possible and depending upon the device risk involved, the device manufacturer should consider auditing the vendor's design and development methodologies used in the construction of the OTS software and should assess the development and validation documentation generated for the OTS software. Such audits can be conducted by the device manufacturer or by a qualified third party. The audit should demonstrate that the vendor's procedures for and results of the verification and validation activities performed the OTS software are appropriate and sufficient for the safety and effectiveness requirements of the medical device to be produced using that software.

Some vendors who are not accustomed to operating in a regulated environment may not have a documented life cycle process that can support the device manufacturer's validation requirement. Other vendors may not permit an audit. Where nec-

essary validation information is not available from the vendor, the device manufacturer will need to perform sufficient system level “black box” testing to establish that the software meets their “user needs and intended uses.

—*General Principles of Software Validation, Section 6.3*

The point of a vendor audit is to establish confidence in the vendor’s quality system and software development and validation capabilities specifically. The level of confidence desired increases with the risk of using the software. A failed audit does not mean that the vendor or the software cannot be used. A failed audit *does* mean that reliance on the vendor’s quality system is no longer a validation tool that will build confidence, so other tools will have to be called upon or the intensity of their use will have to be scaled up to build confidence.

A sample form for conducting and recording a vendor audit are supplied on the DVD that accompanies this text. The questions on the form will help establish whether the vendor does have a software quality system and whether the software has been validated.

Before one invests the time and travel expense on an audit, it is wise to think about the objective of the audit. Is it to qualify or disqualify a potential vendor? Is it to rank confidence in the vendor for comparison to another prospective vendor? Is it simply to find the weak spots so that validation can focus on perceived weaknesses? Knowing the objective ahead of time will help in formulation of questions for the audit that will facilitate the post-audit analysis later.

Conducting an audit that finds numerous problems is valueless unless the device manufacturer does something with that information. What can be done? Additional testing is mentioned above in the GPSV. Verification of outputs may be an option on an ongoing basis. Analysis of known defects with the software is another activity that will allow one to avoid the problems, or to identify process workarounds that can be put in place to minimize the effects of the problems.

### **Known Issue Analysis**

This tool is a valuable validation tool even if the software vendor passed the vendor audit with flying colors. Many suppliers now post defect lists or knowledge bases about their software products on the Internet. Routine, documented analysis of this information periodically reassures one that the software is not failing for other users for the functionality that is important for the medical device manufacturer’s intended use. If defects are found to be reported for critical functionality, then control measures must be established for avoiding that problem, making up for it outside the software, or for repairing damaged information or product resulting from the defect.

### **Safety in Numbers**

A large number of users for a software tool or software-based instrument can build confidence in that software, especially when coupled with some vendor qualification and analysis of known issues. This is *not* to say that no other validation is needed simply because a software item has a large user base. However, one cannot

ignore the logic that a user base of thousands or possibly millions of users who use the software every day are very likely to have already identified most of the defects for the most common usage patterns. What is the likelihood that testing ALT-FILE-OPEN will discover any new defect? Very close to zero. So why spend resource testing that functionality? That kind of rote testing is expensive and time-consuming. It probably will not find defects, and does not build confidence in the software. Researching to find out how many users there are for a given software item at the version level to be used does build some confidence in the stability of the common usage patterns. Researching the known defect data from the vendor for defects in functionality critical to the intended use is a much better use of resources.

For example, suppose a spreadsheet calculates the average and standard deviation for a large table of process control sensor readings. Obviously the AVERAGE and STDEV (Excel) functions would be critical. Should they be tested? Would a test be more likely to find a defect than checking for defects from Microsoft's knowledge base?

Occasionally the intended use will exercise the software in an unusual way, or in a very sophisticated way that only a small subset of the large user base is likely to have tried. Those usage patterns should be identified and tested. Suppose, in the above example, that the table of sensor readings contains some erroneous readings that were replaced with “\*\*\*” instead of the expected numeric value. How do the AVERAGE and STDEV functions react to a “\*\*\*” in the column of values? Is Excel's (probably undocumented) handling of that unusual usage pattern what one would expect or does it affect the results unexpectedly?

Using the safety in numbers argument is not a validation wild card. It is a tool that can be used to focus those valuable test resources where they are most likely to find problems, and free them from the bland, unproductive testing of every feature to be used. In the example, the safety in numbers argument could lead us to conclude that broad general testing of AVERAGE and STDEV is unlikely to find defects. However testing for the “\*\*\*” issue, which is specific to the intended use of Excel for the specific spreadsheet, does add value to the validation. It could identify a problem with Excel, or at least it will clarify its handling of the unexpected input so that results can be properly interpreted, or the data collection process modified to better handle the erroneous values. That is added value from validation, and it came from critical thinking, not rote testing.

### Third-Party Validation

This one is easy...maybe. Occasionally, one can find third-party validation packages for software that is commonly used in regulated industries. Compilers sometimes have third-party validations, as do statistical software tools. One can make use of these when available and if affordable. Some software vendors supply their own validation kits, sometimes for free and sometimes for an additional charge. Vendor-supplied validation is not as common as one might like, but it does seem like one of the best sustainable models for validation of nondevice software.

As much as one would like, even third-party validation is not a silver bullet. It can be very effective, and possibly even better than one could do oneself. However,

there are also many that are practically useless rote tests of the most common functions. Before relying totally on a third-party validation, examine it carefully to establish a confidence level that the validation (which is usually just testing) is truly challenging the software, not just mildly exercising it. Think about the intended use for the software, and evaluate whether the validation adequately covers the functionality critical to the intended use. If not, the off-the-shelf-validation may have to be supplemented with more targeted testing.

One final reminder related to third-party validation: validation is more than testing. Even if an off-the-shelf-validation is used, the other fundamental validation building blocks should be included in the full validation package for the software.

## Output Verification

One advantage nondevice software validation has over device software validation is that often one has control over the conditions under which the software is used. A consequence of this is that the results or outputs of the software can be verified under the controlled conditions to determine if the output is correct; that is, whether the software worked. The verification of outputs can be used as a substitute for at least some of the software testing one might consider for nondevice software. In fact, if 100% of the outputs of a software automated process are inspected for acceptability, that is an ongoing verification of the process which is at least partially automated by software. Wouldn't that be better than a one-time snapshot validation test?

The GPSV gives some leeway in the acceptability of output verification as a validation activity:

Documented requirements and risk analysis of the automated process help to define the scope of the evidence needed to show that the software is validated for its intended use. For example, an automated milling machine may require very little testing if the device manufacturer can show that the output of the operation is subsequently fully verified against the specification before release.

—General Principles of Software Validation, Section 6.1

Note carefully the first sentence of the guidance. What is meant by “Documented requirements and risk analysis help to define the scope of the evidence needed”? Not all output verifications would be adequate substitutes for validation. Suppose that a part created by the software-controlled milling machine had several critical dimensions or tolerances that, if wrong, could result in harm to the patient. If the output verification was a quality control inspection that simply verified that the milling machine created the *correct* part (i.e., it is capable of creating many parts), then that would not be adequate verification of the output to assure that a safety critical failure had not occurred. The risk analysis would identify the critical feature of the part to be created, but the output verification must check for proper implementation of those features to be a meaningful part of the validation of the software.

Output verification is a powerful tool in the nondevice validation toolbox that in many cases is a better solution than snapshot testing of the software. However as with any tool, it must be used carefully and thoughtfully to produce best results.

It would be incorrect to state that validation is not needed because the outputs are being verified. Why?

- Validation *is* necessary and output verification *is only part of* that validation. Output verification cannot be all of the validation because one needs to understand the intended use, requirements, and risks to know what is being verified in those inspections. All of those are validation activities. Further, to protect from the evolving usage of the nondevice software, its configuration and version must be monitored to assure that new or changed features have not allowed those intended uses and requirements to change.
- A simple statement that validation is not necessary, or is fulfilled by output verification leaves no objective evidence for reviewers, auditors, or inspectors to reach the same conclusion objectively. Output verification may adequately replace most *software testing*, but to reach that conclusion must be supported by other validation activities. If that is done, document it! Leave the trail of objective evidence.

The validity and applicability of output verification is not limited to software-driven machine tools that create physical parts and pieces. It applies equally to information producing or information controlling applications as well. Take as an example a CAD system used to design those parts created by the milling machine of the above example. Testing of the CAD software for general use is a daunting and nearly impossible for the user of the software. Most of the above alternative tools could be applied to the validation of the CAD software (in addition to the fundamental tools). Output verification too may be applicable. Identification of those elements of the part design that will be critical to the operation of the device, and verification that they were correctly represented on the output of the CAD system is a form of output verification that supplements the other alternative tools to substitute for detailed rote testing.

One final note on this subject is to point out that there is some relationship in the above examples between the software automated milling machine and the CAD system that designs the parts. The critical features of the parts are determined in design, checked in CAD outputs, and those critical features then are checked on the final produced parts. Thinking about the validation of the two software items independently may not lead to same solutions. These two software items are part of a larger process, each automating part of that process. Understanding the process and the software's role in the process as well as its relationships to other software automating other parts of the process often leads to better validation results for the intended use.

### **Backup, Recovery, and Contingency Planning**

Tools like output verification may be problematic or may not be practical at all for some information management system software. One problem is determining how one would inspect a large database to verify that it is meeting its intended use. That

is not an impossible task, but takes careful design and planning. A second problem is determining what to do if a problem is discovered. With verification of production parts, when a problem or defect is discovered, the part is reworked or destroyed. There generally is not an equivalent solution for information-based systems.

When the intended use, requirements, and risk analysis conclude that the data is critical to the safety of the patients or regulatory record keeping, loss or corruption of data is a serious problem. Scrapping a database and starting over is not really an option. Rote testing may build confidence somewhat, but what if a serious defect first emerges after several million records have been trusted to the software database? Building confidence in this kind of software may require ongoing system or process activities. These are validation activities in the sense that they build confidence in the software. They can be broken down as:

### Proactive Validation Activities

- Design or selection of systems that allow for recovery of databases from archives and/or transaction logs;
- Periodic archiving of data to support a recovery process;
- Periodic data integrity testing to detect possible problems with the data;
- Mechanisms for user reporting of suspected data problems;
- Contingency planning for loss of electronic data or loss of access to software (i.e., what will you do if the server or disk crashes, or while the data is being recovered?);
- Design of recovery processes from archives and/or transaction logs (of course these would be tested before they are needed, right?).
- Execution of contingency plans
- Execution of recovery plans
- Alerting users of loss of data integrity
- Improvement of integrity tests to detect previously undetected defects

This class of alternative validation tools does include some testing, but it is testing of the data itself, not of the software that creates the data. Hopefully it can be appreciated that these activities would appreciably increase the level of confidence in the software, even if a massive software test program were also undertaken.

### Security Measures

Nondevice software in the hands of unintended users can lead to significant safety issues or catastrophic loss or damage to information. Software that allows such unauthorized and unintended uses could be considered defective. Perhaps it won't be considered defective before a loss of data (e.g., due to inexperienced or malicious users), but it probably will be after such a loss! Sufficient security and access control measures help to control these kinds of risks. Sometimes these features are part of the software selected or designed, but the users do not utilize them. An external procedure needs to be in place that requires the use of security measures, and compliance with the procedure needs to be monitored if the data truly is critical.

## Training

Security measures are a good gatekeeper to keep the inexperienced or other unintended users from potentially damaging the data in information-based systems. However, somehow a user will eventually have to become qualified to be considered an authorized user. Training is a solution to this problem.

Training as a risk control measure or a validation activity can be overused. It is too easy to use training as a way to avoid dealing with more difficult human factors issues that would allow (or even encourage) users to err in such a way that data could be damaged or other outputs made defective. All too often, training is used as a generic solution to too many problems or as a cure-all to prevent too many problems.

Those who identify training as an alternative validation tool need to specify exactly what topic in the training will increase the confidence in software if its users have mastered the topic. That is the only traceable way to assure that the training covers all the topics needed to address the concerns about the software. Even then, training as a risk control or validation activity is fleeting. People forget or never understood the training. They change jobs or don't encounter rare issues they were trained on for a long period of time. They may not even remember they were trained on the solution. Once initial installation training is over, new users are introduced to the software by coworkers and may never be properly trained. The half-life of training may be measured in days, but often the training is intended to protect from software misuse for years. That is a problem with overusing training as a solution. It is among the weakest of alternative validation activities, although sometimes there are simply no other alternatives.

## The Validation Plan

The validation plan for nondevice software best considers at least the fundamentals of Figure 16.2 whether it is a do-it-yourself validation or a sophisticated validation of in-house developed software. The likely source of the software will allow one to determine the life cycle model for the software. With a known life cycle model, one can now begin assigning on a phase by phase basis which validation activities should be conducted throughout the life cycle. Articulation of intended uses and requirements belong in early phases of the life cycle. Risk management itself is likely to have activities at almost every phase of the life cycle. Testing and training activities will appear somewhere near the end of the deployment-related phases and perhaps throughout the maintenance phase (if a maintenance phase is appropriate for the software).

There is no single right or wrong way to do this. However some are more right and more wrong than others. A good way to sharpen one's critical thinking about nondevice software validation is get some experience or to look over a number of examples of other's critical thinking. AAMI's Technical Information Report TIR36:2007 provides 14 such examples. These examples were prepared by various individuals in the workgroup that wrote the TIR, so they are representative of alternative ways of approaching nondevice software validation.

The “toolbox” covered in this chapter is a list of some common alternative validation tools. Annex A of TIR36:2007 provides a more comprehensive enumeration of tools, including more traditional validation tools applied to nondevice software. Validation professionals, those who must validate the software they use, and those (like me) who simply don’t trust software, will, with experience, discover their own tools. They will develop favorite tools and methods with which they are most comfortable and which provide them with the most comfort in building confidence in the software they must ultimately trust.

## Reference

- [1] AAMI TIR36: 2007, *Validation of Software for Regulated Processes*.



# Intended Use and the Requirements for Fulfilling Intended Use

## Introduction

Recall that the basic regulatory requirement under 21 CFR 820.70 (i) is to “validate computer software for its intended use.” The previous two chapters have covered the differences between device software validation and the validation of nondevice software. Alternative tools for validation of nondevice software also have been covered. However, there has not been a discussion of “intended use.” If validating for intended use, then intended use should be well understood—so that is it clear whether the validation is on target!

Unfortunately, regulatory guidance in the GPSV is not as thorough on the topic of intended use as it is on some other topics, even though the concept of intended use is mentioned dozens of times.. The GPSV coverage combines intended use with requirements and suggests that intended use be included as part of the requirements specification. The GPSV recommendations that come close to intended use include:

### DEFINED USER REQUIREMENTS

A very important key to software validation is a documented user requirements specification that defines:

- the “intended use” of the software or automated equipment; and
- the extent to which the device manufacturer is dependent upon that software or equipment for production of a quality medical device. The device manufacturer (user) needs to define the expected operating environment including any required hardware and software configurations, software versions, utilities, etc.

The user also needs to:

- document requirements for system performance, quality, error handling, startup, shutdown, security, etc.;
- identify any safety related functions or features, such as sensors, alarms, interlocks, logical processing steps, or command sequences; and
- define objective criteria for determining acceptable performance.

—*General Principles of Software Validation*, Section 6.2

This is not as helpful as it could be, so let us try to parse it a bit to differentiate between intended use and requirements. Additionally, this chapter will provide some guidelines for the types of information that would go into a statement of intended use, and the information that may be more appropriate for requirements specification for nondevice software.

## Intended Use

Once again we find ourselves in a situation in which we all “kind of know” what this important term means. However, upon closer introspection, we come to realize that we may not know *exactly* what it means. Let us attempt to supplement the regulatory guidance to more clearly define “intended use.” One may or may not agree with what this text proposes for the contents of the statement of intended use, but at least it will be a clear, definitive basis upon which the reader can make modifications as appropriate to his or her understanding, opinions, or situation.

### Why It Is Necessary to State Intended Use

Let us start with the question, “Why do we need a statement of intended use?” There are several good answers to this question:

- The regulatory requirement is for validation for intended use. If one is to make the case that software is validated for intended use, then there should be a clear definition of what the intended use is.
- The statement of intended use defines the scope of use of the software. The intended use should be defined as narrowly as possible to fit all of the intended uses that are anticipated.
- Since the intended use defines the scope of use of the software, it also defines the scope of the validation work that will be required to build the confidence and create the objective evidence that the software is fit for its intended use. A narrower intended use allows more focused validation activities.

The following hypothetical examples show how a thoughtful statement of intended use can be used to narrow the scope of validation and focus validation activities on those elements of the software that are most critical to the intended use.

Suppose a manufacturing process engineer wants to use Excel to calculate the average and standard deviation of a certain process variable. The engineer, faced with validating Excel as well as his own spreadsheet program, takes on the task of validating Excel first. He might state that the intended use of Excel is:

“A general calculation and spreadsheet tool.”

Validation for that intended use should be the job of Microsoft, not the process engineer who really has a very narrow intended use for this particular application. Alternatively, he might state as part of his statement of intended use that the intended use of Excel is:

"To calculate the results of a spreadsheet used to compute the average and standard deviation of the process variable X used on the manufacturing line for product Y."

Validation for this intended use is significantly reduced. From this statement, and some knowledge of the product Y and the criticality of the process variable X, one can begin to have a meaningful analysis of the criticality of the correct operation of Excel is to the specific intended use. Further, if our process engineer elects to test Excel, his testing is limited to the AVERAGE and STDEV functions of Excel. If instead he elects to leverage the experience of millions of users of Excel by checking the Microsoft web site to see if there are any reported problems with Excel, he can narrow his research to those two particular functions.

### **Intended Use and Validation of Nondevice Software**

Intended use can be the friend of those charged with validation of nondevice software because it does narrow and focus validation activities to those specific areas of functionality that are critical to the intended use. One must be cognizant, however, that this type of validation is only applicable to the intended use as stated. Furthering the preceding example, imagine that in later improvements to the spreadsheet, the engineer decides to extend his spreadsheet to organize his process variable calculations for a month's worth of data. Further, he would like to graphically display both the average and standard deviation throughout the month as the data is collected. Not only has his spreadsheet changed considerably, but his use of Excel too has changed. The previous validation of Excel (or his spreadsheet) would not apply to his new intended use.

Before continuing on with the discussion of intended use, note that this approach to validation of generic software tools like Excel is considerably different from a generic validation of Excel for any and *all* intended uses. The latter approach was the accepted norm until several years ago. Since the release of TIR36:2007 and the emphasis on critical thinking for validation of nondevice software, many have made validation of generic tools specific for each particular intended use, deemphasizing rote testing of the general tool, which was unlikely to find any defects or build any confidence. Instead, validation emphasis is placed on the spreadsheet or other inputs or outputs associated with the generic tool, which are more likely sources of error.

### **Contents of a Statement of Intended Use**

The intended use is the equivalent of the product requirements or system requirements that were discussed for device software. Intended use is the highest level description of what function the software is going to serve. Many struggle over how much or how little goes into a statement of intended use. There is no one quantitative answer that would apply to all nondevice software items. If, however, one understands the purpose(s) of the statement of intended use, then the expected length of the statement of intended use is easily determined. It is simply long enough to adequately achieve its purpose by describing each anticipated use for the software item.

What is meant by the purpose? Recall in Chapter 10 the description of the system requirements specification (SyRS) for a medical device. It was a description of all the functions and features of the device that were required to fulfill the users' or other stakeholders' needs. Design validation for devices requires that it "shall ensure that devices conform to defined user needs and intended uses" {21 CFR 820.30 (g)}. Those user needs and intended uses are described and documented in the SyRS or similar document.

What is needed for *nondevice* software is a description of the functions and features that will be required of the software to meet stakeholder needs—the equivalent of an SyRS. The regulation that applies to *nondevice* software {21 CFR 820.70 (i)} requires that the device manufacturer "validate ... for its intended use." To know whether the validation is thorough enough or whether it is complete, it should be traceable to a documented intended use.

Where does the information in a statement of intended use come from? The same place the contents of the SyRS come from—the users and other stakeholders. The *purpose* of the statement of intended is threefold:

1. It establishes a goal for validation. It defines what *exactly* the software has to do to meet the needs of the users and other stakeholders. It defines an end point for validation. The software must fulfill the intended use, but validating more than that is not required by regulation or guidance.
2. It defines a goal or objective for those who will either design the software or who will identify and select the software if it is off-the-shelf software that will be acquired for use. If the nondevice software is to be custom developed, then the statement of intended use actually is an SyRS.
3. It defines a context for the use of software item for determining the risk of use of the software. Management of the risk is closely related to the validation of the software.

## Determining Intended Use

How does one determine the intended use? The intended use for a medical device is usually related to a diagnostic or therapeutic use, and ancillary needs from other stakeholders. What is the equivalent of the diagnostic or therapeutic intended use for nondevice software? Recall that the nondevice software validation that is required under 820.70 is for software that automates part of the production or quality system. Therefore, the intended use of the software first describes that part of the production or quality system that it automates.

Many find it useful to diagram the process of which the software is a part, especially if it is part of a complex production process. This is an excellent way to gain insight into the interfaces that the software has with other parts of the process. This is useful not only in understanding the intended use of the software, but also in the context of the risk management process for nondevice software. An understanding of the whole process can make more apparent the risks of using the software in the context of the process, and may also be useful in identifying where other risk control measures may exist within a process to protect from software failure.

Once the role of the software is established in its surrounding process, the specifics of the intended use must be established.

### What Exactly Is the Software Supposed To Do?

This is not a key-press-by-key-press description, but once the role of the software in the larger process has been established, a description is needed of the functions the software should provide to fill that role. The boundary between this functional description of intended use and the requirements (covered later in this chapter) is not distinct. However, being too brief or even evasive in the description of what the software is to do does not allow for meaningful risk analysis, validation planning, or confidence building activities. If there is uncertainty about the type or amount of information that should be provided in a statement of intended use, one should simply ask oneself whether there is sufficient information to itemize the major functions of the software to be tested (or developed for that matter).

### Who Are the Intended Users of the Software?

In an earlier chapter the relevance of the *number* of users was discussed at length, so that surely should be part of the statement of intended use. The experience level of the intended users is important. It is important to know if they are assumed to be well experienced and likely to understand the terminology and user instructions the software may provide, or if they are expected to be novice users. The expected training level of the intended users is just as important. One should know whether a formal training program is expected to be required before allowing a user to use the software, or whether it should be intuitive to a new untrained user. The language or languages of the intended users is important to know for designing, selecting, and testing the software. Anything else about the intended users that may be relevant to designing or testing the software or otherwise managing the risk of its use should be noted.

### Where the Software Is To Be Used Can Also Be a Relevant Intended Use Factor

Ambient light conditions can be a factor in the choice of colors to use for backgrounds and fonts on the screen. Human factors issues like how far the user is expected to be from the user interface can affect font sizes. The intended use environment can determine the state of mind of the user. For example, if the software is to be used in a high-stress environment, or in situations that demand a short response time, that is important to note as part of the intended use. Why? Because the software should be validation tested in expected use environment, to be sure it will be effective when deployed there. The ultimate users of the software will not be impressed that it passed all the tests in the lab if they cannot use it on the shop floor. Where the software is to be used geographically should be considered, especially if the manufacturer is to use the software item in a number of locations worldwide. Location can affect not only the default language, but the units of measure, time and date formatting, and so forth, that, if not properly designed, configured, and tested could result in confusion of the user that may result in hazardous situations. Even

cultural differences have been known to affect the way a user interacts with software, and therefore should be taken into consideration.

### When Is the Software To Be Used?

Yes, even this can be important for accurately assessing risk, control measures, and other validation activities. In this case, *when* usually means when in the process.

#### *Example: Spreadsheet*

Imagine a spreadsheet that calculates the number of revolutions per minute required of a motor to pump at a specified delivery rate for an infusion pump. Now consider three different points in time in the development life cycle that the spreadsheet might be used:

- During early hardware specification phases, the spreadsheet is used to calculate maximum RPMs needed to specify a motor that runs at the maximum rate.
- During software implementation, the spreadsheet is used to calculate RPM rates for all operating speeds of the pump. Those values are hard-coded into the device software.
- During validation testing of the device that ultimately is designed, the validation team decides to check RPM values of the motor in a verification test. They use the spreadsheet to calculate expected values for their test designs.

For the first of these uses, one might argue that if the spreadsheet is wrong, the wrong motor would be purchased. Later testing of the final device would detect that the motor could not reach maximum speed. After all, one would certainly test the maximum speed, right...right? Certainly the risk is reasonably low.

However, in the second case, suppose a large number of operating speeds are supported by the device. If a wrong RPM value were calculated by the spreadsheet, it may be for a pump speed that would not specifically be tested in the final device. There is a somewhat higher risk if the spreadsheet is used for this purpose. Calculated values become part of the device, and it may not be obvious that they are wrong if there is a failure of the spreadsheet.

That brings up the third situation. The spreadsheet is used to compute test values for validation testing. *Now* if the values are wrong, the test would be wrong. That could result in a false positive test failure (i.e., test failed, but software really is not defective). That in itself would be of no concern if subsequent analysis recognized it as a false positive, fixed the test value, reran and passed the test. A false *negative* result (the test passes, but there really was a failure) of the validation test due to an incorrectly calculated value usually would be considered to be very improbable. It is very unlikely that one would just happen incorrectly to calculate the expected value to be the same incorrect value in the software—unless—the software designer and software tester used the same flawed spreadsheet to calculate those values!

There are several points to be made with this example. First, the risk profile of a piece of software is almost entirely determined by its intended use. Second, the intended use is composed of several categories of information that define that

intended use, including when and how it is used in the process that it is automating. Third, there sometimes can be synchronized canceling failures of software that go undetected. In the above example the same spreadsheet could have had two intended uses: (1) calculating speeds for software implementation, and (2) calculating speeds for test development. It failed, but since it was used and did not reflect failure in the same way for both uses, the failures canceled each other and the test failed. The test failed in a false negative way. A problem existed in the device that was not detected because a defective software tool provided the same wrong answers to the development and test team!

Had this been recognized, perhaps the spreadsheet would have been subjected to more rigorous testing, or perhaps it would have been determined unwise to use the spreadsheet as a cross-check of itself in this way.

Intended use (and requirements) do matter. Too often they are not taken seriously and are short, vague, almost titles of a usage instead of well thought out descriptions of as many aspects of the usage as possible. Unfortunately, this has a detrimental effect on rest of the validation process that depends on the statement of intended use and the requirements details.

## Requirements for Fulfilling the Intended Use

If all of the above was to describe the intended use, then one might ask what is left for requirements. It is bad form to define things in negatives, but:

1. Requirements are not simply repeats of the intended use written in requirements language.
2. Requirements do not come from attempts to reverse engineer out what one would have imagined a software requirements specification (SRS) would have looked like (for software that has been acquired instead of custom developed).

## Requirements for Custom-Developed Software

Requirements for custom-developed software may, and probably should be, written like a formal SRS. Why would it be right to have SRS-level requirements for custom-developed software, but not for off-the-shelf or otherwise acquired software (which will simply be referred to as “acquired software” for this topic)? It comes down to the purpose for the requirements. That is, why do we need requirements?

In the case of the custom-developed software, the requirements define the goals for the subsequent software design and ultimate validation. They also are the basis for the system-level software verification tests. It is the first time the software has existed, so detailed requirements and a formality of development and validation process will lead to better software in which one has a higher level of confidence.

Requirements for custom-developed software should be at much the same level as for *device* software for all the same reasons. They lead to well-written software in which the users have a high level of confidence. No more will be said in this chapter

about those requirements. One can refer to the chapter on requirements for device software for more details.

### Requirements for Acquired Software

Now, let's focus on requirements for acquired software. Software that is acquired for use in the device manufacturing organization that is subject to the 820.70(i) regulation probably represents more than 80% of the nondevice software anyway, not counting spreadsheets. (That 80% estimate is not based on anything factual, just from personal experience and observation.)

In the case of acquired software, it's already too late for requirements to serve as a goal for design, implementation, or verification testing. Hopefully, the supplier of the acquired software has already done that. So there is very little value in reverse engineering requirements, other than testing those requirements. Reverse engineered requirements come from the observed behavior of the software, so of course the verification tests will pass regardless of whether the software works as intended! The only defects that will be detected from such an exercise are defects noticed by reverse engineering or defects in the test procedures themselves. This is not a value adding activity, and is quite accurately perceived as wasting time.

For acquired software, one purpose for requirements is to guide the selection process for acquiring software that will meet those requirements. However, if requirements are reverse engineered from the software, is that purpose fulfilled? Just the fact that the “engineering is reversed” implies that the software was acquired *before* the requirements for it had been determined. So there is little or no value in the requirements activity even for product selection. Is there any other purpose or value at all?

There is a second purpose for determining requirements for acquired software—even if the software is acquired first. That is, oddly, to determine whether the software meets the needs for the intended use through verification. Requirements analysis can be, and almost always is, an iterative process. Requirements can be collected from other requirements-generating activities such as risk analysis and risk management. Risk control measures can often generate requirements for the software to control a certain risk. If the acquired software does not or cannot meet those needs, then requirements for other procedural control measures (outside the software) will be needed. Note that the identification of these requirements *before* the software is acquired affords one the opportunity to select software that satisfies the requirements. Identification of the requirements *after* the software has been acquired does not eliminate the need for the requirements. It simply puts one in the position of deciding to do without the requirements (since the software cannot change), or to satisfy the requirement through other external procedural means.

### Information Content of Requirements

Earlier, it was confessed that there is a murky line between the kinds of information in statements of intended use and requirements. There is really little use in having that debate, but two guidelines that have proven themselves valuable in making this determination are:

### Requirements Are Verified; Intended Use Is Validated

Recall that two components of intended use to be considered are the whos, wheres, and whens (i.e., who uses the software, where it is used, and when it is used in the process). One would not write a verification test for an intended user (verify that the user is an English-speaking production engineer), but one *would* verify the requirements for language by verifying that every message is appropriately translated into the necessary languages for the intended users. One would then *validate* that those intended users are successful in using the software in the intended use environment and that the software is effective in automating the process where it is used in the overall process.

### Requirements Support Intended Uses

This is best explained in a series of examples:

1. In addition to describing the functions it automates, a spreadsheet's statement of intended use reveals that it is to be used by a number of users, some of whom may be poorly trained and inexperienced with spreadsheets in general. In response to those details of intended use, the spreadsheet designer determines that the calculated fields should be locked to prevent accidental (or malicious) alteration of the formulas embedded in the spreadsheet. That is a requirement to support the details of the intended use.
2. A software system is acquired. Part of its intended use is to collect device master record (DMR) related data that must be approved by the production supervisor electronically within the system. Subsequent analysis reveals that this use will make the software subject to the Part 11 regulations on electronic records and electronic signatures. All of the requirements of the Part 11 regulation now apply to the software. These regulatory requirements are *requirements* that support the intended use of the software which in part manages regulated information in a regulated way.
3. A PLC that will control a production process describes in its statement of intended use the details of what it is controlling and the specifications and tolerances required for that control along with the intended users, intended use environment, and so forth. Upon further analysis, it is determined that there is a specific startup requirement for the process controlled by the PLC, and a specific shutdown requirement to safely shut down the process. The startup and shutdown sequences are *requirements* that support the intended use of controlling the process.

Requirements come from a formal requirements analysis processes, or as mentioned above, they may originate from risk analysis and management activities. Examples might include requirements for alerting or alarming, or input range or validity checks, or other interlocks that would prevent a user from accidentally misusing the system in an unsafe manner.

Requirements also can come from experience with the software, or may be discovered subsequent to encountering a defect or other problem with the software. New requirements can be spawned by new features introduced by updated versions

of the software, or by evolving intended uses for the software. As mentioned earlier, it is often an iterative process.

Requirements often are quantitative measures or other objective measures of the performance of the software for its intended use. For example, a machine vision automated inspection system may have as one of its intended uses that it will “inspect ABC parts to identify and reject X and Y defects.” Some of the performance and throughput requirements to support that intended use may include:

- The X and Y defects shall be detected with a 1% or lower false positive rate and a 0.01% or lower false negative rate.
- The inspection system shall be capable of inspecting a minimum of 1,000 parts per hour for both defects.

Other types of requirements may include requirements for

- Access control, user authentication, levels of security
  - Subordinate requirements for managing the security settings
- Encryption of data
- Descriptions of safe states (for software-driven machine tools)
  - When the software should have the machine in a safe state
  - Any required sequence for getting into or out of a safe state
- Handling power failure or premature shutdown of the software
- Recovering from power failure or abnormal shutdown
- Data import or export
- Data integrity checking
- Recovery from corrupted data.

As with so many other itemizations in this text, the above list is not meant to be a comprehensive list, but an example of the kinds of things that may be considered in a requirements document for acquired software. It is meant as a few examples to stimulate creative thinking to identify requirements appropriate for one’s particular application.

### **Example: Intended Use and Requirements for Validation of a Text Editor**

This chapter concludes with an example of a validation of a simple text editor from our office. There was a proliferation of text editors in use in the office. Every software engineer had his or her favorite, and some engineers had several that they used. Few (i.e., none) were validated. In the interest of promoting the sustainable model of having users validate their own software, the engineers were asked validate all editors they intended to continue using. The first benefit of this policy was that the software engineers rather quickly came to the conclusion that they could “settle” for using one text editor company-wide. They selected one editor and took on the task of validating it.

The usual complaints were aired that such a general purpose tool could “never be validated,” but they were encouraged to continue nonetheless. We reviewed the six fundamental validation ingredients for DIY validation. The process began.

The first statement of intended use was something like:

The XYZ text editor will be used to create ASCII text files.

That was a little disappointing but when prodded the software engineers were able to articulate a little better what their intended use was:

The XYZ text editor will be used to create, edit, and format ASCII text files that will be used as software source files and make files that will be subsequently post-processed and checked for syntax errors, and whose executable versions of the software will be subject to their own validation.

That is only a little longer, but significantly narrows the scope of what would be needed to validate it. In particular, the phrase “subsequently post-processed and checked ... for errors” and the fact that the executable software that would result from the source code would also be subject to some verification and validation made this validation look like one that could rely at least somewhat on verification of 100% of the outputs of the editor.

The team charged with validating the tool researched the supplier of the tool and found that it had a large user base, but came from an individual in Eastern Europe. They reselected a different editor that met their requirements, had a larger user base, and that was supplied by a domestic company that had been in business for years and would more likely be around to support the tool. Vendor selection became part of the validation. There was safety in numbers with an installed base in the hundreds of thousands of users.

There was a full accounting of known defects on the supplier web site for the version they were using. They read through the known errors, determined that only a few of them could potentially affect their use. A small training document was created for new and existing users that warned of the potential defects. The user base reporting of defects was leveraged as test results.

There were a few advanced features the team identified as requirements for the editor, and they naturally were interested in checking that those features worked the way they were expected to work. This resulted in a few targeted tests.

The fact that the source files were always post-processed made the team comfortable in making their case (through a risk management process) that any testing other than the advanced feature testing would be unlikely to be productive. A configuration management plan was created for controlling installations of patches or new versions as part of the maintenance phase of the life cycle.

The validation report was several pages long. It was divided into sections with headings for the six fundamentals, but the thought process that lead them to the validation result was very natural. They had actually thought of many of the issues that they addressed in the validation report before they were asked to validate, but had not documented them. However, what was new included:

- The supplier research surprised them and caused them to choose another tool.
- They recognized and documented the limited scope for which the text editor was validated.

- They identified defects reported by other users that could affect their work, and warned other coworkers through a training document.
- They systematically thought about advanced features they used that might not be common usage among the user base. They recognized that their confidence was somewhat lower for these features and decided to do some testing. No defects were found, but their confidence was enhanced.
- *They recognized that what they expected to be impossible and a waste of time had actually been a valuable exercise!*

Several weeks after the validation was completed, one of the engineers recognized that there was another intended use for the text editor. A unit test tool we use creates the test results in an ASCII text stream that was captured in a text file. The text editor was used to automate scanning of the file for the text that flagged an erroneous result (\*\*ERROR\*\*).

The validation package was updated to include this additional intended use. It seemed like such a simple difference it was hardly worth the change. As part of updating each of the sections, someone realized that although it would be the same SEARCH function of the text editor that would be used to search for \*\*ERROR\*\* that was used routinely to search while editing a source file, there was a difference. The unit test result files could be very large (hundreds of thousands of lines). This spawned a requirement that the editor, and specifically the SEARCH function, had to work on files up to 500,000 lines long. This seemed like an unusual usage of the tool. Consequently, confidence was shaken and a test was devised to be sure the editor would find all instances of \*\*ERROR\*\* throughout a 500,000 line file. As luck would have it, the editor did *not* pass the test. The editor did not work beyond 65,535 lines ( $2^{16}$  lines).

The net effect was that a task that was initially presumed to be a waste of time actually produced good results, increased the level of confidence in the software, and was accomplished in a reasonably short amount of time of focused effort. Understanding intended use allowed the task to be narrowed to a manageable size, and the recognition of a change in intended use lead to the discovery of a defect that would not have been an issue with the original intended use.

The tool was *not* validated for all conceivable uses (that should have been the job of the supplier). It *was* validated for the specific intended uses. That is the responsibility of the device manufacturer, and is all the device manufacturer should be interested in anyway.

# Risk Management and Configuration Management of Nondevice Software: Activities that Span the Life Cycle

In Part II of this text, several activities were discussed that span the life cycle of the device software. Those activities included risk management, configuration management, planning, reviews, and defect management. With the exception of risk management, these activities differ little for custom developed *nondevice* software for which the full range of controls are available. Configuration management for nondevice software, which is acquired for use, does differ somewhat from configuration management for software that is custom-developed. This chapter will focus on the differences in risk management and configuration management for nondevice software. The reader is trusted to return to Part II to review the fundamentals of planning activities, reviewing outputs, and managing defects. The minor differences as they apply to nondevice software should be apparent and will not be discussed further in this chapter.

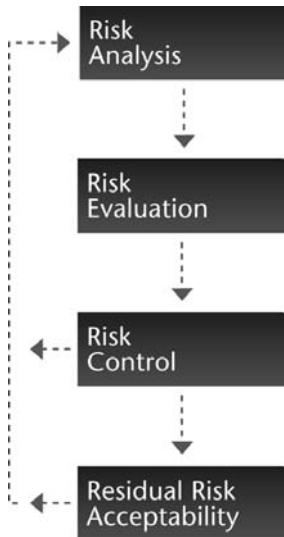
## Risk Management

The risk management activities for nondevice software are quite different from those that were described for device software. The risk management process steps are quite similar, but the details of the harms, probabilities, and available control measures are significantly different and deserve separate coverage.

As with device software, risk management and validation for nondevice software are very closely related. Most of the validation activities that boost confidence in the software could just as validly be considered risk control measures that reduce the risk of harm from use of the software. The ISO 14971 standard on the application of risk management to medical devices considers device software as a source of risk in medical devices, but does not consider nondevice software. More recently, the AAMI 80002 guidance on the application of ISO 14971 to medical device software targets risk management for medical device software, but again makes little mention of the management of risk from software used as part of the design, development, production, or other parts of the quality system. As with 14971, the basic processes are applicable, but the details are significantly different.

### Applying the 14971 Risk Management Process to Nondevice Software

The risk management process for medical devices that is described in the ISO 14971 standard does not apply directly to nondevice software. Figure 18.1 is repeated here



**Figure 18.1** The ISO 14971 risk management process.

from the chapter on risk management for device software to show the major process steps of:

- Risk analysis;
- Risk evaluation;
- Risk control;
- Assessment of acceptability of residual risk (evaluation).

The simple process represented by the flowchart to identify risk, and subsequently to control it down to an acceptable level is totally applicable to nondevice software. The language we will use to describe risk for nondevice software is the same. The relationships among the terms are the same. Risk is a combination of the severity of a harm and the probability of that harm. Hazards are sources of harm, and there are causes for those hazards. Beyond these similarities is where the differences begin to emerge.

## Harm

ISO 14971 concerns itself primarily with the risk of harm to patients or caregivers. Nondevice software risk management covers a broader cross-section of risk and must consider:

- The patient;
- The caregivers;
- The risk of noncompliance with regulations;
- The production personnel who may be harmed by the software;
- The risk of harm to the environment;
- The risk of harm to the business.

It is not entirely clear where the regulatory requirement for risk management ends and common sense begins. However, it would make little sense to consider the safety of patients and ignore the safety of one's own coworkers.

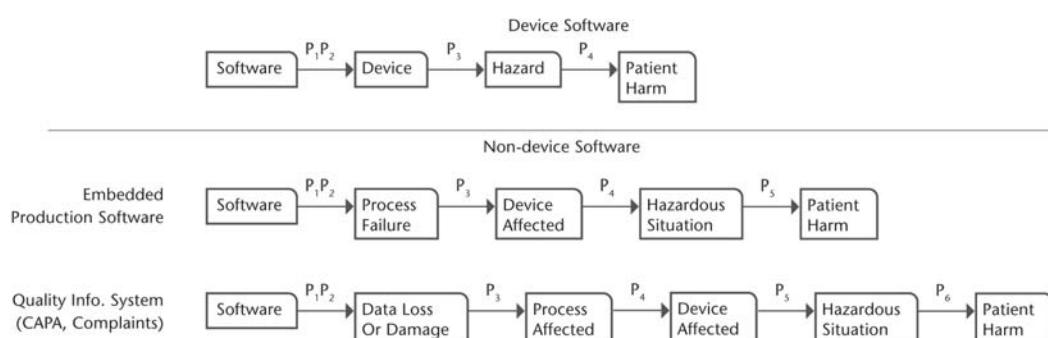
The regulatory requirement {21 CFR 820.70 (i)} for validation of nondevice software introduces a new concept for risk and harm: regulatory risk. Take for example the regulatory requirement for validation of electronic record or electronic signature (ERES) software. Failure of an ERES system that tracks training records and signatures *may* remotely and indirectly be involved in some failure that *could* result in patient harm, but it *directly* could result in a regulatory finding of noncompliance if records of compliance were lost—thus a “regulatory risk.”

Failure of software embedded in a production tool could result in harm to the operator and bystanders. Other production software failures could result in spills or accidental venting of toxic materials that could be harmful to the environment. Software failures that result in production shutdowns or recalled product could be harmful to the business financially, by reputation or by regulatory response.

Failure of a *device* could directly harm a patient or user, whether the failure is related to failed parts, production errors, use error, or design flaws. For example, if a pump overdelivers a drug therapy, the patient could be injured. If it underdelivers a pain medication, the patient may be subjected to unnecessary pain. However, failure of software alone never directly injures a patient or clinician. The actual injury is delivered by the device or process in which the software is embedded. In the above example of the infusion pump harms, it is the pump *system* that over- or underdoses. The embedded software may miscalculate the dosage or may have timing issues that make the delivery inaccurate, but without the surrounding electromechanical system, the software errors would result in nothing. Harm from device software is a cause of a device system hazard and ultimately risk of harm.

In the case of *nondevice* software, a failure of the software is even one more level removed from harm to the patient. Figure 18.2 shows this difference graphically for failure of several examples of nondevice software.

Although failure of nondevice software is even farther removed from harm to a patient than a failure of device software, nondevice software failure would be just as direct a path to harm to production workers, to the environment, and to the state of regulatory compliance as a failure of *device* software would be to patients.



**Figure 18.2** Relationship of software failure to harm.

### Risk, Severity, and Probability

Recall that risk is a combination of severity of harm and the probability of that harm. There is not much to add here about severity (at least as far as severity of injury is concerned) that wasn't said in Chapter 8 on device software risk management. However, the fact that nondevice software risk management could consider six or more types of harm (listed above), some of which are *not* related to safety does pose some interesting questions. If one were to rank the risks by severity alone, would regulatory risks rank higher than risks of injury or at least some risks of injury? We would all agree that the risk of serious injury or death to a patient or production worker would rank higher than the risk of a 483 inspectional finding from the FDA. Would the risk of a nonserious injury to the patient rank below the risk of a 483?

There is not a good answer to this question, but it calls attention to the issue to keep risk management and validation in perspective by the industry and by regulators. Ultimately, regulatory risks should relate to safety. Regulatory requirements exist for good reasons. Recall that the primary mission of the FDA is to assure devices are safe and effective. All of the regulatory requirements, some of which might be automated by software, are there (or should be) to promote safety or effectiveness.

To better clarify the question of severity for regulatory risk, consider a nondevice software system that tracks the users of an implantable device so that they can be notified in the case of a recall. Figure 18.3 shows a map of how the failure of the tracking software could result in harm to a patient. The probabilities of failures in chains of events such as these multiply. The probability that the tracking software fails in such a way that *some* device(s) cannot be tracked is  $(P_1 \times P_2 \times P_3)$ ; however, just because a device is not tracked does not mean that harm is a certainty. First, there has to be a device failure (probability  $P_a$ ) that would result in a recall (probability  $P_b$ ) and that the failed device(s) are in the group that cannot be tracked due to the software failure (probability  $P_c$ ). Together then, the probability that the tracking software fails in a way that damages or loses the tracking information, *and* there is a failure of devices that result in recall and those devices are in the group whose records are damaged is:

$$(P_1 \times P_2 \times P_3) \times (P_a \times P_b \times P_c)$$

Then there is the probability that the suspected failing device actually is failing or would fail ( $P_x$ ) and the probability that the failure would result in harm to the patient ( $P_y$ ). Overall, the probability of harm occurring to a patient because of a failure of the tracking software is:

$$(P_1 \times P_2 \times P_3) \times (P_a \times P_b \times P_c) \times (P_x \times P_y)$$

Now consider the regulatory risk of failure of the software. For probability of this, one need only consider the probability that the software fails ( $P_1$ ), in a way that makes the device manufacturer non-compliant ( $P_2$ ), and is detected by a regulatory inspector, or is self-detected and reported to the regulatory agencies ( $P_z$ ). Therefore the overall probability of a regulatory risk materializing is  $(P_1 \times P_2 \times P_z)$ .

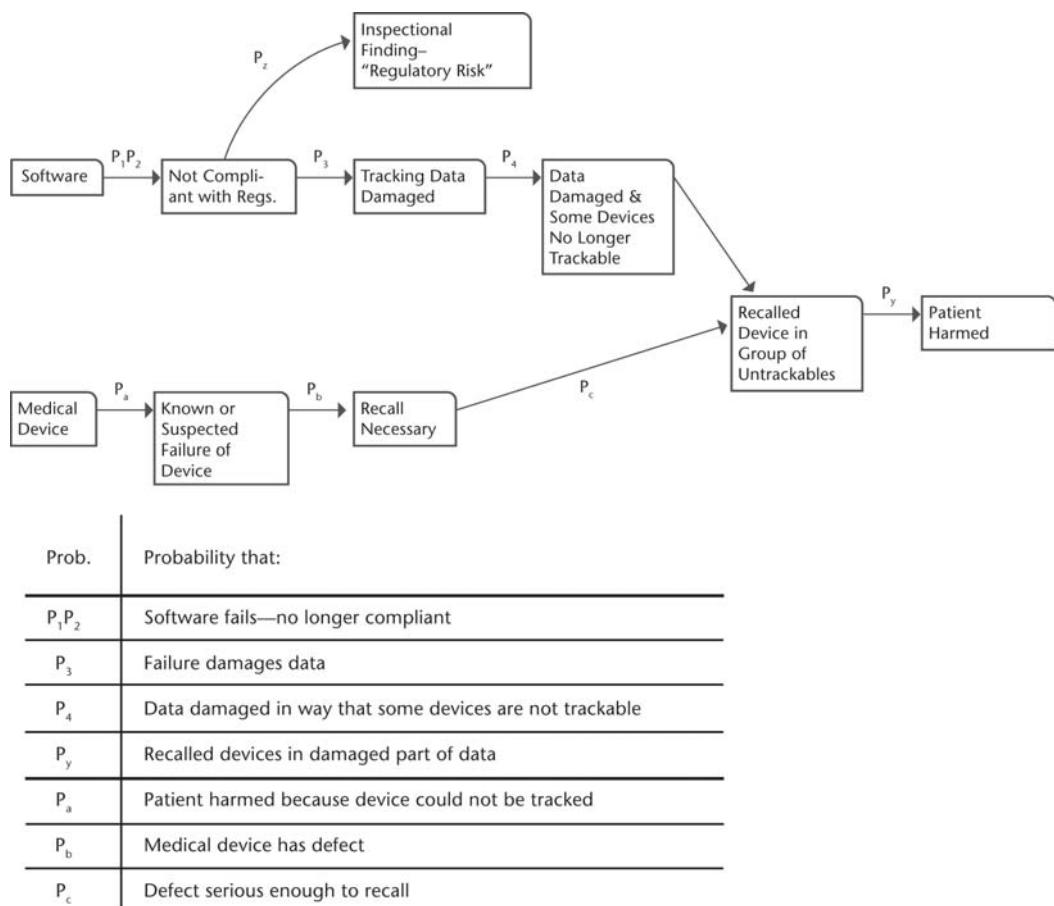


Figure 18.3 regulatory risk maps to safety risk.

Of course, the probability of software failure is difficult to estimate, as are all of the other probabilities in these calculations. Some of the probabilities are fairly remote, and some are almost certainties for certain situations. For example, if the software failure destroys all data, the probability ( $P_c$ ) that a failing device's data is lost is equal to 1. Even though the real probabilities are difficult to estimate, let us assume several scenarios in which all probabilities are equal to make a point.

Table 18.1 shows the odds (reciprocal of probability) of a regulatory risk and a safety risk for the tracking software example if it is assumed that each probability in the chain leading up to the “harms” is assumed to be an equal value.

**Table 18.1** Example Odds of Harm ( $P^1$ ) for Regulatory Risk and Safety Risk

Probabilities	Odds of Regulatory Risk	Odds of Safety Risk
50%	8	256
10%	1,000	100,000,000
5%	8,000	25,600,000,000

The table shows that if each probability were estimated at 10%, the odds of a regulatory risk are one in 1,000. However, the odds of a safety risk are one in one hundred million! The odds of a regulatory risk being realized in this example are a hundred thousand times more likely than a safety event.

That is why the question of *severity* of regulatory risk and business risk is an important one to consider. If risk is quantified as the product of probability and severity, and we assume that the “severity” of a regulatory risk and a safety risk are equal, then a risk-based validation for this example would be a hundred thousand times more focused on eliminating the regulatory risk than the safety risk. That is probably not the right thing to do, and it is probably not what the regulatory agency would want to have done.

Device manufacturers often have a narrow focus on reducing regulatory risk as much as possible. They probably would agree, without any probability calculations, that they are more likely to suffer “regulatory harm” than a user is likely to suffer any physical harm. The result of focusing on regulatory risk for our product-tracking software example might lead the device manufacturer to spend inordinate amounts of effort on testing the software to “be sure” it will not fail. Volumes of test results often are produced to convince regulatory inspectors that the intent of the regulation has been met. This is not a case of maliciously doing the wrong thing. In fact, device manufacturers often spend more than they need to in an attempt to do the right thing.

Of course those of us who do not trust software know that one cannot “be sure” that software will never fail. So, when the software in our example does fail, the device manufacturer is surprised and may not have planned what to do if the software ever did fail.

What is the right thing to do with severity of failure of nondevice software? Regulations, guidance documents, and standards do not address this issue. Although some documents including TIR36:2007 mention the concept of regulatory risk, managing regulatory risk outside the context of managing safety risk leads to behavior that is contradictory to the overall goal of the regulations. Perhaps it is better to think about how failure to meet a regulatory requirement could lead to a safety event. In fact, the risk map of Figure 18.1 shows how meeting the regulatory requirements for validation {21 CFR 820.70 (i)} and ERES (Part 11) fits in with the overall risk tree. If one assumes that the regulatory requirements are enforced logically, then satisfying the goal of reduction of risk of physical harm will, as a by-product, also satisfy the regulatory intent and thus reduce the regulatory risk.

To estimate the probability of harm for the example in Table 18.2, all probabilities were set to the same value. One might be curious how the probability of harm total would be affected if the individual probabilities were set to “realistic” values. Two scenarios of “realistic values” were used for the calculations shown in Table 18.2. These two scenarios use very conservatively high probability estimates for each step of the process. Even so, the overall odds of the patient being harmed due to a failure of the device tracking software are between 1 in 3,200 and 1 in 20,000,000. Note that these odds are before any control measures or additional validation activities have been added. The point again is that with many levels of indirection between the source of failure in nondevice software and harm to a patient, the odds of such a failure resulting in the patient harm are not large.

**Table 18.2** Two Estimate Scenarios for Probability of Harm

<i>Probability</i>	<i>Probability That</i>	<i>Scenario 1</i>	<i>Scenario 2</i>
$P_1 P_2$	Software fails—not compliant	50%	50%
$P_3$	Software failure damages database	50%	10%
$P_4$	Data damaged—some devices are not trackable	25%	1%
$P_a$	Medical device has a defect	50%	50%
$P_b$	Defect serious enough to recall devices	20%	20%
$P_c$	Recalled device records in damaged part of data	10%	1%
$P_y$	Patient harmed because device could not be tracked	50%	10%
Probability		0.0003125	0.00000005
Odds (1/P)		3,200	20,000,000

Before we leave the topic of probability, the odds in Table 18.1 and Table 18.2 are intriguing enough to linger on the topic a little more. In several contexts elsewhere in this text, reference has been made to risk factors that can qualitatively reduce or increase the risk associated with a given item of software. One of those risk factors was the number of events that needed to occur simultaneously or in sequence for a hazardous situation to result in harm. It has been proposed that the larger the number of events needed, the lower the probability the situation would result in harm. Table 18.3 shows the odds of a sequence of events resulting in harm, assuming each event is independent and has an equal probability of occurrence.

As one can see from the table, the odds of a nondevice software failure resulting in harm become extremely remote for even moderately long sequences with relatively conservative probability estimates. For example, a chain of only five events has only at most a one in 3.2 million chance of resulting in harm if each event has a 5% (that is, 1 in 20) chance of occurrence. These tables cannot be used to read off the probabilities for any risk analysis; they are based on rather simple assumptions. However, they are good for showing just how sensitive the overall probability is to the number of events that must occur for harm to be realized. In the example just cited, the odds of harm increase to one chance in 1.28 billion if only two more events are in the chain of events leading to harm.

This kind of probability analysis applies to risk management for devices and device software as well. However, for nondevice software the results are more striking because harm to a patient is more indirect than it is for device software.

**Table 18.3** Odds of Harm Given Probability and Number of Events

<i>Probability of Each Event</i>	<i>Number of Independent Sequential Events Leading to Harm</i>					
	2	3	4	5	7	10
50%	4	8	16	32	128	1,024
25%	16	64	256	1,024	16,384	1,048,576
10%	100	1,000	10,000	100,000	10,000,000	10,000,000,000
5%	400	8,000	160,000	3,200,000	1,280,000,000	1.02E+13
1%	10,000	1,000,000	100,000,000	10,000,000,000	1.00E+14	1.00E+20

## Managing the Risk

The preceding treatment of probability not only puts the risk of harm for nondevice software in perspective, it makes it more clear how risk control measures and other validation activities can quickly reduce the probability of harm resulting from a software failure down to very acceptable levels.

In the example from Figure 18.3, let's consider several risk control measures that could be added to reduce the risk even further. Three to consider, with their probabilities of failure, are:

1. Periodic backups of the database to protect from total loss of data ( $P_q$ );
2. Data integrity checks that test the data periodically for corruption ( $P_r$ );
3. Collection of transaction logs that record the raw transactional activities so the data could be reconstructed ( $P_s$ ).

All are good common sense best practices, and probably without a risk analysis, many would reflexively decide to protect the data in at least one of these ways.

How do they affect the probability of a software failure resulting in harm? The pathway in Figure 18.3 that was labeled  $P_4$  was the probability that the software damaged the data in such a way that a device was no longer trackable. If one only implemented the first control measure above, the probability of not being able to track a device would become:

$$P_4 \times P_q$$

and if all three control measures were implemented, the probability of the software failure resulting in harm would be:

$$P_4 \times P_q \times P_r \times P_s$$

In this case it would take failure of all three control measures to lose the data for tracking a device. For even modest values for  $P_q$ ,  $P_r$ , and  $P_s$  of 1%, the probability of harm occurring is reduced by a million to one (1% x 1% x 1%). Even if  $P_1$ ,  $P_2$ ,  $P_3$ , and  $P_4$  were assumed to be 50% each (one chance in 16 of harm from Table 18.3), when all three control measures are applied with estimated probabilities of failure of 1% each the odds of harm becomes 1 in 16 million!

Unfortunately, as attractive as these numbers are, we still cannot estimate the probabilities of failure accurately. However, over time, experience with the software may allow some refinement of the estimates, which may also infer how many control measures should be applied in the process and where. Admittedly, the loose way probabilities have been used in these examples is not totally valid. The actual dependencies and independencies should be clearly understood if one is going to try to make a quantitative argument about probabilities of risk. The main point of the discussion here is that external control measures can drastically reduce the probability of error resulting in harm. The million-to-one improvement made possible with the three control measures above would be hard to duplicate with only testing of the nondevice software as a control measure.

Risk management is part of validation, and certainly confidence is boosted by externally applied control measures as done in the example above. A good

validation plan will test those control measures to be sure they are effective. Further, those results should be part of the validation package for the nondevice software being validated.

### Controlling the Process to Reduce Risk

Risk management for nondevice software also differs from risk management for device software in the opportunities available for controlling risk. In the case of device software, one must anticipate potential hazards and failures that can be the cause of those hazards. Additionally, any control measures for those risks must be designed *into* the device system since there is little control over the device or its use once it leaves the factory floor.

In the case of device software, more opportunities are available for risk control measures because most of the processes that are automated or partially automated by the software are still within the confines and control of the device manufacturer. Some of the alternative methods of validation that were described earlier are external methods for controlling the risk related to nondevice software failure. For example, the three control measures that were mentioned for the device-tracking software example above are reasonable because they are within the control of the device manufacturer. If, however, the very same software were produced by *the device manufacturer* to sell to hospitals for tracking their own device implant histories, reliance on the external activities of periodic archiving, integrity checks, and transaction logging would be a less effective risk control measure because there is no guarantee that the customer hospital would actually implement the three external control measures. They would be weaker control measures simply because of the probability that they would not be implemented correctly, or at all.

In Chapter 17, on the topic of intended use, it was recommended that the process (of which the nondevice software is a part) should be diagrammed or otherwise fully understood to determine the role of that software in the process. The same recommendation holds for risk management. Unless the overall process that includes the software is fully understood, the risks cannot be adequately estimated and effective control measures may be overlooked.

### Risk Acceptability

The risk management process of ISO 14971 that was reviewed earlier in this chapter included an assessment of the acceptability of residual risk after control measures have been applied. In the chapter on risk management for device software, the question of *exactly* what would constitute acceptable risk was sidestepped. At that time the activity simply was defined as comparing the residual risk to the definition of acceptable risk in the risk management plan. That may seem now like a trick, but there is plenty of precedence for that kind of explanation. ISO 14971 essentially says the same thing, but takes a little longer to say it.

The challenge to this is defining the risk acceptability criteria. Before considering some sample definitions, let us consider a process for defining acceptability criteria and attributes for good definitions.

1. Define risk acceptability early in the life cycle. When software is custom-developed, defining acceptability in the risk management plan is ideal. When managing risk gets difficult, it is a natural human tendency to lower the standard of acceptability to meet schedules, budgets, or just to make it easier. Committing to a definition of acceptable risk in writing keeps one honest. The written definition does not migrate. Of course, it is possible that the original definition is too ambitious or unachievable and has to be changed. Often that is fine, or even necessary, but at least a written record of the migration of the standard of acceptability is documented this way. Somehow that seems to keep “acceptability inflation” under control.
2. When defining the acceptability criteria, provide the logic or argument for why those criteria are appropriate for the risks for that specific device. This helps later generations of maintenance engineers to understand the rationale, and reminds the risk management team of the rationale should a change in the acceptability criteria be considered later in the life cycle.
3. Acceptability criteria should be written and judged in the same way requirements are. They should be unambiguous, precise, consistent, and testable. One should be able to judge whether a residual risk is acceptable or not *objectively*. In other words, the results of the decision should be understandable by a reasonable reviewer without the aid of verbal explanations or interpretation. Implicit in this suggestion is that imprecise language is discouraged, including “as low as reasonably practicable” which is (I know) used in Annex D of the ISO 14971 standard. Unfortunately, language like that, although it seems practical, will have a different meaning to every person called upon to make the evaluation of whether the residual risk meets that criterion.

With these goals in mind, let us examine how one might go about defining risk acceptability criteria that meet some of the above goals. The first thing to point out is that one of the attractions to quantitative risk analysis is that it makes the setting of acceptability criteria and the evaluation of meeting the criteria much simpler. The first decision one must make is whether risks will be scored quantitatively or qualitatively.

### Quantitative Risk Acceptability Criteria

Quantitative risk measures are challenging for all the reasons that have been mentioned a number of times in the discussions of risk management for nondevice software as well as for medical device software. The three major problems with quantitative scoring are:

1. Determination of quantitative probability estimates or software failures and other failures of the process in the chain of events leading to harm;
2. Determination of appropriate quantitative severity scores that will appropriately scale the responses to the perceived risk.
3. Determination of the formula for combining the probability score with the severity score to determine risk.

Enough has been said about probability already, so further treatment will not be given here. The above discussion on probability does give a framework for attempting to use probabilities in a quantitative way without having to depend on arguments of whether the probability of software failure is one in 100,000 or one in 1,000,000. Overall probabilities in those ranges may result from analysis that deals with a series of probabilities in ranges that are much easier to understand.

Assuming the combination of probability and severity that yields a risk score is linear, then scoring of severity deserves some deliberation to assure that the score definitions will scale the risk estimate appropriately. In other words, if one assumes an equal probability for two different risk scenarios with different severities, will the risk scores adequately represent the perceived risk and the expected response to control that risk?

For example, assume that one risk could result in the harm of a minor injury, and a second risk could result in death. The probabilities are equal, so presumably the risk score will be solely determined by severity. If the severity scale runs from a quantitative value of 1 to 5, and the severities for our example are assigned values of 2 and 5, in the risk score for our death scenario would only be 2.5 times higher than that of the minor injury scenario. That is unlikely to result in appropriate scaling of control measures to manage the risk of death compared to that of the risk of minor injury. If risk is defined as the product of the severity score and the probability score, then the risk of a minor injury that is only 2.5 times more likely than the risk of death would result in the same risk score. That does not seem to pass the common sense test.

If purely quantitative scoring is to be attempted, one way to determine what severity scale should be used is to compare severity levels two at a time. Then ask, “At what relative probability levels for these two severity harms would I be accepting the risk with no further control?” Consider a minor injury harm of severity  $S_1$ , and a harm of permanent injury of severity score  $S_2$ . To find the values of these severities relative to each other, one might conclude that probability of occurrence of the minor injury of 1% is acceptable. Similarly, one might conclude that the acceptable probability of occurrence of the permanent injury,  $S_2$ , is 0.001% (1 in 100,000). (This is not to say that these are appropriate probability values for *any conceivable* software. Determination of the threshold needs to be the responsibility of the device manufacturer.) Now, let us assume that our risk score is simply equal to the product of the probability and severity for each risk. The risk of acceptability threshold then is defined as  $S_1 \times 1\%$ , which would be the same as the threshold  $S_2 \times 0.001\%$  (threshold is threshold, after all). Since the two threshold scores are equal, one gets that:

$$S_2 \times 0.001\% = S_1 \times 1\% \quad \text{or}$$

$$S_2 = 1000 \times S_1$$

That is, the severity of permanent injury would need to be 1,000 times the severity of minor injury to allow straight quantitative scoring to be used for risk acceptability determination.

### Qualitative Risk Acceptability Criteria

Chapter 8 covered a number of qualitative methods for evaluating risk acceptability in devices, including rule-based and relative comparison methods. Those will not be repeated here; they are explained adequately in Chapter 8. One other method that is worth mentioning is the Safety Assurance Case method, which is gaining some momentum recently in safety standards, in FDA guidance documents [1], and in public forums in which the FDA participates.

So far, assurance case methodology has only been mentioned by regulators in the context of device risk management, but there is no reason it could not be used just as effectively for nondevice software risk management. It is especially effective for qualitative arguments because it provides a format for explaining rationale.

### Safety Assurance Case Method

There is a growing literature base on assurance case methodology. There will not be adequate space here to go into assurance case methods in detail, but it is important to cover it enough to point interested readers in the direction of getting more information if it is of interest.

Assurance cases can be organized in one of two ways: the claim, argument, evidence (CAE) or goal structuring notation (GSN) methods. For our purposes here, let's focus on the CAE method. This method makes a claim, presents an argument to support the claim, and points to evidence that corroborates the facts that are leveraged in the argument. Of course, it can get more complicated with subclaims, subarguments, and so forth. For our purposes, the discussion will be restricted to just one level of claim and argument.

Our safety claim may be something like:

The XYZ software is safe to use for the intended use specified in the ABC document.

The argument(s) to support the claim may be of the form:

- The risk of Hazard A is acceptable because it is of severity level X.
- The risk of Hazard A is acceptable because it requires a chain of at least Y events, the probability of which is estimated to be less than Z.
- The risk of Hazard C is acceptable because the Control Measure 3 reduces the severity to security level X.
- The risk of Hazard C is acceptable because the Control Measure 4 would increase the number of simultaneous (or chained) events that result in harm to Y events. The probability of this is estimated to be less than Z.

Any *argument* that is made in support of a claim is itself supported by *evidence* that the argument is true. In the above examples of arguments, the evidence would be documented analysis that determined the revised severity or probability estimates, or some experimental evidence that the arguments are true. In the case of risk control measures that have been added, the evidence would include a reference to the new requirement for that control measure, and the verification results for proving that the control measure was implemented and effective.

Both the CAE and GSN methods for assurance case development rely on graphic methods for communicating the logic. The difficulty of analyzing and communicating the overall risk evaluation required by the 14971 standard is aided by this overall graphic method for communicating the rationale for claiming risks have been adequately controlled.

The assurance case methodology is equally applicable to medical device software and nondevice software. It is more prevalent in Europe, and appears to be gaining acceptance in the United States with the FDA. Unfortunately, space will not allow any further coverage of it in this text. The references at the end of this chapter are a good starting point for learning about the methodology. In particular, the Kelly thesis [2] presents an informative coverage of the history of the slightly different notations, and the problems other industries have had in incorporating assurance case methods. Kelly also proposes some extensions to the GSN method to address some of the problems reported by others.

### Detectability

In Chapter 8, related to failure mode analysis of devices and device software, it was mentioned that detectability of a failure or hazard was of limited value in controlling risk *unless* something happened in response to the detection to control the risk. The same is true for nondevice software; detectability of a hazard alone is not sufficient as a control.

With nondevice software there are situations in which the device manufacturer is in more control of the situation as a *user* of the software. Since nondevice software is automating a process or part of a process that is of importance to the device manufacturer, detectability of a failure may be by a trained professional who knows how or is trained how to react, thus making detectability highly desirable. However, it is the reaction of the professional that reduces the probability of harm, not that the failure might be noticed. Of course a trained professional with no way of knowing about a failure, can do nothing about it. It is the combination of detection and correction that reduces risk. Either alone are very weak control measures. Perhaps a better way of thinking about detectability is that without it, risk increases.

## Configuration Management for Nondevice Software

Nondevice software that is custom-developed has most of the same configuration management issues as medical device software does. Keeping software and supporting documentation in synchrony during the development process is a large part of the concern. Configuration management during development will not be revisited here since there is so much in common for between device and nondevice software in development phases.

Differences do emerge for software that is acquired for use. There are also a few differences in configuration management that should be considered once custom-designed nondevice software is put in use.

## Why Configuration Management Is Important

Configuration management (CM) was mentioned as one of the fundamental validation building blocks for nondevice software. One might ask why CM should be such a concern for software that is simply acquired for use or has already been designed for use. Several major factors are in play that should be considered when developing the CM plan for software that is acquired for use. Those factors include:

### *Intended Use*

Chapter 17 described the importance of understanding the intended use of the software for validation (for intended use). In fact, it was a tool that allowed the validation activities to be scoped downward to focus on intended use. It is common for software to evolve over time. New functions and features are added to keep the software competitive. If software is upgraded by the software users at the medical device manufacturer without any control or analysis, it is quite possible that the user will begin using some of the new features, thus changing the intended use of the software. There exists a possibility that previously unanticipated issues could arise if the validation is not updated for new intended uses.

### *Intended Users*

Just as the intended use of the software changes with feature and function changes, so too do the intended users. New functions may attract new users. Those users may have totally different training and experience levels that could introduce problems with the *use* of the software. Other factors also may be introduced by new unanticipated users such as language, units of measure, or cultural habits (such as order of operations) that differ from the originally anticipated user base. A new user base should prompt a reassessment of the validation plan for the software.

### *Intended Use Environment*

The same is true for the intended use environment. New functionality may enable the software to operate in environments that were not previously anticipated. What is meant by environment? Certainly the “computing environment” which, among other concerns, includes communications connectivity that may lead to issues with remote access and control, security issues, and threats from malware.

### *Compatibility*

Compatibility is, perhaps, what most would think of as part of the configuration. One must not be tempted to think about the software in isolation when considering the configuration and how to manage it. The “configuration” includes everything with which the software interfaces, such as:

- *Hardware.* Hardware compatibility includes the hardware upon which the software runs and any particular interface hardware such as driver boards, display electronics, volatile and nonvolatile memory configurations, and user

input devices. Software that controls other equipment or instruments should take into account the version and configuration information for that with which it interfaces.

- *Other software.* Nondevice software that runs on general purpose PCs runs within the confines and control of an operating system, and perhaps even other levels of software. A version change of any of the software with which the subject software (i.e., the software being validated) interfaces could impact the validity of the operation and/or results of the subject software. All interfacing software should be considered as part of the validated configuration of the subject software.
- *Data.* Other users, other instances of the subject software, and other software that shares the same data are all tied together in the “configuration” that must be managed. Anyone who has ever used a word processor or data-based software application such as a requirements management tool or defect tracking system has experienced this issue. One user upgrades his software, which restructures the database. That data is shared with other users whose software is not upgraded and can no longer operate until they are upgraded. Often this kind of issue is simply an annoyance. However, sometimes it can become a serious issue if the data is also accessed by remote users (such as customers, vendors, or regulators). Just as serious is access to archived data if the migration path of old data is not well planned to the new data structures. The user has some control over some of these issues, and no control over others. Careful planning of configuration changes is needed to avoid potentially serious problems.
- *Process.* Nondevice software automates a process or a part of a process. The interface between the software and the process is also something that must be managed and if that interface to the process, or the process itself changes, the validation status of the software must be reassessed. For example, consider a software item that was purchased to automatically inspect finished parts of a medical device coming down a production line. If the production rate is increased because of improvements upstream in the production process, will the inspection software be up to the task of completing its function successfully in the reduced time available? A process change could prompt a reassessment of the state of validation—the configuration has changed.

## Configuration Management Planning

Some of the elements of a configuration for software that automates part of a process cannot be controlled by the user. This is unlike configuration management during the design and development processes of a custom-developed piece of software in which the developers and testers have full control over the various components of the software configuration (of course there are exceptions such as embedded off-the-shelf components). Consequently, a configuration management plan (CMP) must be proactive on those elements of the configuration over which there is control, and plan to be reactive on those elements over which there is no control.

The first part of a CMP is simply a definition of the configuration. As a minimum, the concerns listed in the preceding section of this chapter should be

addressed as shown in Table 18.4. Of course, more critical and creative thinking is always a good idea given the particular circumstances of the subject software.

Note that for many acquired-for-use software items, a number of the elements will be blank. For subject software that documents intended use, requirements, risk management, and test artifacts separately, the CMP need only refer to those documents and their version number that define the configuration for the version of the subject software.

Table 18.4 addresses the definition of the configuration. The CMP, being a plan, is more than definition of the configuration. It should also address the activities that will be incorporated to *manage* the configuration. In other words, what will be done

**Table 18.4** Suggested Elements of a Software Configuration for Acquired-for-Use Software

<i>Configuration Element</i>	<i>Information Needed</i>	<i>Suggested Details</i>
Subject software	Details of the software whose configuration is being managed.	Software name, manufacturer or other source, software version number, update method.
Intended use	Information relating to intended use of sufficient detail to determine if use changes in future.	Reference to document that details this information, and the version number of that document.
Intended users	Information relating to intended users of sufficient detail to determine if user base changes in future.	Reference to document that details this information, and the version number of that document.
Intended use environment	Information relating to intended use environment of sufficient detail to determine if expected environment changes in future.	Reference to document that details this information, and the version number of that document.
Risk Management plan	Information relating to anticipated risks, and identified control measures of sufficient detail to determine if they change significantly in future.	Reference to document that details this information, and the version number of that document.
Hardware platform	Details of hardware upon which the software is run.	Manufacturer, model number, memory configuration, available disk space, interface requirements and version (e.g., USB 2.0).
Hardware interfaces	Details of the instruments, equipment, other electronic to which the device will interface (if any).	Manufacturer, model number, device version, embedded software version, update method.
Other software	Details of operating systems, platform software, other software to which subject software will interface.	Software name, manufacturer, software version number, update method.
Data	Details of data that is shared by subject software with other users or other applications.	Some identifier of the “version” of the data configuration. Most likely this will be a version number of the software that creates and maintains the data (which could be the subject software).
Process	Description or process diagram that details the role of the subject software in the process of sufficient detail to determine if it changes significantly in the future.	Reference to document that details this information, and the version of that document.
Test procedures and results	Information related to any testing that was performed on the software.	Reference to documents that detail this information, and the versions of those documents.

to keep the configuration from changing, what the process will be for changing it if necessary, and what will be done if the software changes unexpectedly? The 3 Rs of planning that were discussed in Chapter 9, the roles, responsibilities, and resources should be considered for each activity that is identified. Using RASCI charts (Chapter 4) is a good method for assigning the 3 Rs to each activity identified as part of the CMP.

### Configuration Management Activities

*Management* implies some action on the part of the manager. For each of the elements of the configuration, the configuration manager (or at least the person creating the CMP) should consider what activities are appropriate for each configuration element. A change in any configuration element should prompt a reevaluation of the state of validation of the software, which includes a reevaluation of the risks associated with the use of the software.

The following questions may lead the configuration management planner to the appropriate CM activities for each element of the configuration:

1. How many versions of the subject software will be permitted if there are to be multiple instances of the software? Should all users be required to use the same version? How will that be managed? How will all the instances be identified, tracked, and located in case of a required update?
2. Who will determine when software (i.e., subject software or any other software in the configuration) changes? How will that decision be made? How will defects be reported? What is the decision-making process for processing defect reports and responding to them? Who will be responsible for deployment of software changes?
3. Is it possible for uncontrolled changes to be made to the subject software or any other software that is part of the configuration? If so, how will they be detected? Whose responsibility is it for monitoring for uncontrolled changes? What is the plan if an uncontrolled change is detected?
4. How will changes to the use/user/environment be detected? Who will be responsible for monitoring this, and what is the appropriate monitoring interval? How will this activity be documented?
5. What happens if the supplier of the software changes or becomes unavailable? What happens if any of the configuration items becomes unavailable for future deployments? What is the plan for dealing with software problems if the supplier goes out of business or is unresponsive?

This probably seems like a lot of overhead at this point. For most simple, low-risk software, many of these configuration items and activities will not apply. For complex situations, the configuration and management of it will be correspondingly complex. In any event, at least considering the above questions will better prepare the medical device manufacturer for problems that can be anticipated.

## References

- [1] *Total Product Life Cycle: Infusion Pump—Premarket Notification [510(k)] Submissions—Draft Guidance*. U.S. Food and Drug Administration, Center for Devices and Radiological Health. April 23, 2010.
- [2] Kelly, T. P, *Arguing Safety—A Systematic Approach to Managing Safety Cases*, University of New York Department of Computer Science, September, 1988.

# Nondevice Testing Activities to Support Validation

## Why Test—Why Not To Test

This may seem like an odd way to begin a chapter on software testing, but in many cases, voluminous testing of nondevice software may not be the best option for returning value on the validation investment. Common sense would indicate that any software that is being used the first time with the first formal users *would* benefit from considerable testing. However, software that is acquired for use and that has a user base of hundreds of thousands of users *probably* is not going to benefit much from systematic reverse engineering and testing by a medical device manufacturer who simply wants to use the software.

Custom-developed software, custom-configured software, custom-modified software, or any other software of limited circulation should be considered for some formal testing using the same processes and procedures that are applicable to custom-developed medical device software as described in Chapter 13. Why? There are two good reasons why:

1. *Because it's custom.* As the first and potentially only user of the software, there is no benefit from earlier users who likely would have already reported, common defects that they experienced. The software should be tested simply because there is no other "safety net" to protect the new users from defects in the software.
2. *Because you can.* As an acquirer of custom software, presumably the needs, intended uses, and requirements for the software were communicated to the developer (hopefully in a documented form). Testing the implementation for compliance with those requirements is more meaningful than testing acquired software for compliance with reverse engineered (imagined) requirements. Further, custom software can be modified if testing reveals that the software does not meet the needs or requirements. Try that with shrink-wrapped software!

Does that mean that noncustom software that is acquired for use does not need to be tested at all? Not exactly. There are some regulatory reasons that will be discussed later that indicate that *some testing* should take place. As earlier chapters of this text have suggested, satisfying regulations should be a by-product of doing the right thing for validation. That attitude is much healthier, logical, and productive than focusing on compliance with regulations and hoping that the software works as a by-product of compliance.

Why would one want to do any testing of large user base acquired-for-use software if it is likely that the most common defects have already been reported and fixed? The answer to this goes back to the definitions of verification and validation. In the context of nondevice software validation, testing could help establish confidence in the software by concluding that:

- The software meets the needs the medical device manufacturer has for the software. That is, it is the *right software*. This is partially accomplished through *validation* testing.
- The software is free from defects. That is, the software *works right*. This is accomplished through *verification* testing, inspection of known defect lists, or faith in the experience of large numbers of prior users.

All nondevice software, whether custom-developed or shrink-wrapped off-the-shelf, benefits from some testing to come to the first of these conclusions: this is the *right software*. That is, it meets the needs.

The second conclusion, that the software *works right*, is best accomplished by detailed verification testing for custom software. However, off-the-shelf software does not come packaged with requirements and design information, so detailed verification testing is limited to whatever can be reverse engineered from the way the software is observed to be working. That is a time-consuming and expensive activity. It does not have much chance of discovering defects. It only accomplishes, in a very expensive way, verifying that the software works the way it works. That does not have much value, and is probably not what the regulators had in mind.

Time out. At the beginning of this Part (Chapter 15) on nondevice software validation, despite a healthy distrust of software it was recommended that one should, “trust ... but verify.” Now, the preceding paragraph sounds like verification testing of off-the-shelf software is a waste of time. Isn’t that a conflict? No. Verification can be accomplished in other ways than the reverse engineering method. When verification testing does not make sense (because the software is acquired for use), there are other substitute methods, such as:

- Verification that other users are not experiencing issues with functionality important to your intended use;
- Verification of software outputs as part of the process of using the software;
- Verification that the supplier followed a good development and validation processes.

All are verification/validation activities that build confidence in the software. For software that is acquired for use they are less expensive and are more effective than reverse engineering.

Custom software should have more of the full range of verification and validation activities as possibilities, and they should be exercised accordingly. Since testing of custom-designed software is so much like that of device software, the reader is referred to Chapter 13, which deals with device software testing. The rest of this chapter will be more focused on the acquired-for-use types of software.

## Testing as a Risk Control Measure

Often, in reviewing risk management documents or FMEA analyses, the control measure or mitigation that is cited is “software testing.” That is, a failure mode or risk that is attributable to a software failure is controlled by testing. In some cases, testing may be the only possible thing that can be done. However, testing is a *weak* risk control measure. Software testing is an imperfect activity just as software development is imperfect. Testing for software that is acquired for use is even less perfect. For highly critical, safety-related functions, testing is a good idea, but testing alone may not be sufficient to provide ample confidence in the software. Validation of such critical functionality should refer back to some of the alternative methods in the validation tool box to support the test results. More imperfect methods are better than fewer.

## Regulatory Realities

One must realize that in an agency the size of the FDA or any other regulatory agency, the inspectors will have a spectrum of levels of experience, training, understanding, and opinion of what the regulations say and how the guidance documents are interpreted. From the viewpoint of the medical device manufacturer, this can lead to what sometimes seems to be random responses and opinions coming from regulators who inspect software validation artifacts.

In a well-publicized court case in which the United States (i.e., the FDA) sued Utah Medical Products Inc., one of the major issues had to do with validation of software that was embedded in a piece of production equipment. Utah Medical claimed that the medical devices produced by the equipment were all inspected and tested, thus providing 100% verification of the output of the software-driven machine. The FDA called witnesses who claimed that validation of such production software *always* was comprised of the minimum components of requirements, testing, and a test report.

One logically can come to the conclusion that testing software may be less effective than alternative methods discussed in Part III of this text. In fact, one could easily come to Utah Medical’s conclusion that testing would really add little to the confidence they had in the output of the software. However, the FDA, or at least some within the FDA, had the opinion that testing was *always* necessary.

In very few (if any) instances in this text have activities been proposed strictly to meet regulatory intent that were not also beneficial to building confidence in the software. This may be one of those exceptions. Although the FDA lost the suit, we now know that some within the FDA may feel strongly about software testing. As a defensive regulatory strategy, device manufacturers should include *some testing* in a validation package regardless of the logic that might lead one to conclude that no testing is necessary. This is not to suggest frivolous testing just to have some testing to show an inspector. If resources are to be spent testing, they should be spent wisely by testing where the results are most important, are the most likely to find errors, or both. The techniques described below will guide the tester through this process. The testing does not have to be excessive, just thoughtful. The testing should show that

the requirements for the software have been successfully fulfilled by the software. Even though the testing may be unnecessary from a quality and safety perspective, it may still be productive in finding defects that could be otherwise damaging to the manufacturer.

## Testing Software That Is Acquired for Use

When software that is acquired for use is tested, there are some things that can be done to maximize the odds of finding critical defects, thereby maximizing the return on the testing investment.

First, focus the test effort on the most critical functionality. The risk management results should be a guide for which functions of the software the automated process is the most sensitive to. These functions may not be the most likely to have defects (although, sadly, they often are), but they will be the defects that are the most important to find.

Second, test around areas of functionality that have to do with the particular intended use for the software. Obviously, it makes little sense to test functionality that is not to be used. In particular, focus testing on functionality related to the intended use that may be used in an unusual way. In other words, if the anticipated use will stress the software in any way, those unusual usages make ideal candidates for testing. Stressing the software might include storage capacity, speed, length of records, or unusual query or report requirements. Recall the example of the text editor validation in Chapter 17. The testers were concerned about the editor's search function for a string in the unusually large text files for a certain intended use, and found a defect (actually an undocumented limitation) in the software.

Third, look for defects in those areas of the software in which they are most likely to be found. How are you supposed to know that? In the discussion of the qualitative probability of software failure in Chapter 8, a long table of situations or conditions were listed that made individual software functions relatively more likely to fail. Many of those listed would not apply to software that is acquired for use because they rely on knowledge and control of the development process. However, some that do apply include:

- *Functionality of high complexity.* Prospective users know what functionality is complex without the aid of a design specification. Often the complexity of a task is what attracted the user to the software in the first place. Complex *anything* is more likely to fail than the less complex—and that includes software.
- *Functionality that is least documented or understood.* Users of software are often left wondering how certain functions of the software work, or what certain labels or messages in the software mean because the user documentation is lean. If one is not sure how the software is supposed to work, then some exploratory testing is appropriate to find out how it works and to discover the limitations of the software. Exploratory testing often generates valuable information on the undocumented limitations of software that are not, technically, defects or failures. Communicating those limitations to other prospective users of the software can save heartache later.

- *Functionality that is used in unintended ways.* Occasionally an off-the-shelf software package will be used for ways that were not anticipated by the developers. For example, in our office, we have used our requirements management tool for managing our risk management file and for managing our defects. There is no reason to believe the tool will not work for our additional intended uses, but since there is some probability that our “off-label” use might stress the software in a way not anticipated by the developers, there is value in doing some testing around the new intended use before relying on the software for that purpose.
- *Functionality that has a history of defects.* Several times in this text it has been mentioned that software that has had a problem in the past has a higher than average probability of having problems in the future. When one has full control over the development of the software, this information is readily available, but how can one make use of this in testing off-the-shelf software? Software suppliers that make defect databases available offer a glimpse of the history of the software after it was released to the market. User groups and other experience-sharing web sites also provide an understanding of where troubles may be hiding in the software. Note that even though a problem has been reported and fixed by the software supplier, that area of functionality should be considered suspect and a target for some testing because it has a higher likelihood of having other problems than areas of functionality with no historical complaints logged against it.

Once the decision is made to test, and a plan is in place for what functionality will be tested, then the question is how to test the software. Software that is acquired for use has limited options for testing. For the most part, only system-level (black box) testing applies to this kind of software. The system-level software testing techniques of Chapter 13 apply equally to software that is acquired, except that the requirements will be limited to those that are determined by the prospective user to meet the needs for the intended use. In particular, boundary value testing, performance testing, stress testing, error guessing, and normal and abnormal input testing should all be considered. Testing may be enhanced by mixing scripted and unscripted (use case) test procedures and by using experienced and inexperienced testers.

## IQ, OQ, and PQ Testing

Validation of off-the-shelf software and equipment for laboratory and production use is often divided into three phases: installation qualification (IQ), operational qualification (OQ), and performance qualification (PQ). Validation of nondevice software is not required to follow the IQ/OQ/PQ structure of testing, in fact, the FDA’s *General Principles of Software Validation* (GPSV) only mentions IQ/OQ/PQ in passing and does not use the terminology or further refer to the methodology in the guidance. The GPSV is careful to say that the IQ/OQ/PQ terminology that is familiar to process validation professionals who work within the production and laboratory environments is not as familiar to software professionals who were the

main audience of the GPSV. If thinking of validation in terms of IQ/OQ/PQ is more familiar or more applicable to the acquisition method for the software, then there is no reason not to use it. Conversely, if the acquisition method or intended use of the software is not a good fit for the IQ/OQ/PQ terminology, there should be no reason to force the use of this terminology.

The IQ/OQ/PQ phases of software testing based on the implicit assumption that the software is acquired for use (off-the-shelf), or was custom-developed and delivered to the production a laboratory environment for IQ/OQ/PQ qualification. The qualification activities during the three IQ/OQ/PQ phases breakdown as follows:

- Installation qualification verifies that the configuration of the software, which presumably is described in the configuration management plan, has been correctly implemented, and that the software has been installed as it was intended to be. This implies that the installation instructions should be available in written form if they become part of the requirements that are being verified during IQ.
- Operational qualification verifies that the software actually operates once it is installed and up and running. OQ test procedures verify that the expected requirements of the software for its intended use are fulfilled in the operating software. OQ testing is verification of the requirements that have been articulated for the software to fulfill its role in automating a process, or part of a process. (The requirements that are verified in OQ testing are the requirements described in Chapter 17, which are typically *not* the detailed software requirements specifications (SRS) that one would use for designing and developing the software. That level of testing would be done during the design and development of the software by the software supplier.) OQ testing may be performed outside the intended production or laboratory environment.
- Performance qualification verifies that the software successfully achieved its intended use by automating the process or part of the process for which it was intended. PQ testing is performed in the intended use environment, within the process in which it was intended to operate, with intended users. PQ testing is similar to validation testing that was described for medical device software testing. PQ testing can be accomplished through scripted test scenarios, semiscripted use case testing, by monitoring the performance of the software in a live environment according to a documented plan, or a combination of all of these methods.

We often see device manufacturers try to use IQ/OQ/PQ terminology to describe the validation of custom-developed production automation software throughout the design, development, and postdelivery phases of the software life cycle. It is not a good fit. IQ/OQ/PQ is a best fit for *after delivery* of the software.

Relatively recently, the term design qualification (DQ) has been used by some. The introduction of DQ was needed to address the problems introduced by trying to force the IQ/OQ/PQ terminology into predelivery phases of the software life cycle. DQ testing might be thought of as the traditional software requirements verification that has been discussed elsewhere in the text that takes place during the development

of the software (which was explicitly *excluded* from the description of operational testing above).

The IQ/OQ/PQ has not been used elsewhere in the text for many of the same reasons that the FDA did not use the terminology in the GPSV. The fact that the terminology has not been used is not an indication that it is not useful. In fact quite the opposite is true. IQ/OQ/PQ fits nicely with the way process engineers think about process validation and allows software validation to fit neatly within that organization. Where it is appropriate, IQ/OQ/PQ is a great tool. When IQ/OQ/PQ is forced into scenarios where it is not a good fit it becomes problematic.

## Validation of Part 11 Regulated Software

The regulatory requirement in Part 11 regulations for validation of software that is used to automate electronic records and electronic signatures was presented in Chapter 15. In that presentation of material it was suggested that Part 11 validation was *not* a validation requirement *in addition* to the validation requirement of 21 CFR 820.70(i). There were some peculiarities and specifics that went along with Part 11 validation that should be covered at this time.

Part 11 contains 36 (by my count) requirements for electronic records and electronic signatures (ERES) software. Validation of ERES software involves validating the Part 11 requirements for those parts of the ERES software that are part of the intended use. For instance, if one chooses not to use the electronic signature feature of an ERES software application, the Part 11 requirements for electronic signatures would not have to be validated.

The six validation fundamentals would apply to ERES software that is acquired for use along the guidelines that have been presented in Part III of this text. One peculiarity is that the risk management activity should, in addition to safety, consider the risk of documentation loss, corruption, unauthorized access, misuse, malicious intentional damage, and the risk of regulatory action if required documents are destroyed or damaged.

The 36 requirements can be verified by testing, which sometimes simply boils down to inspection of the software for the existence of a required feature. Alternative methods such as checking with the supplier and online user group databases for known problems with functionality required by Part 11 are applicable for building confidence.

Since the topic of this chapter is testing, and in light of the FDA realities mentioned above, a conservative, defensive strategy for validation of Part 11 compliant software probably should include some testing since inspectors have two regulatory reasons (Part 11 and 21 CFR 820.70i) to question the validation of the ERES software. An organizational strategy that makes it easily apparent to an inspector that the Part 11 validation requirement has been met is to separate the Part 11 validation from the other functional validation of the ERES system and unambiguously trace each of the applicable 36 Part 11 requirements to a verification test (or inspection procedure). A table of the 36 Part 11 requirements is supplied on the accompanying DVD to facilitate this process.

## Summary

Testing of nondevice software that is acquired for use often is not the most productive activity in building confidence in the software. However, sometimes it is the *only* control measure for risks associated with failure of a critical function of the software. Additionally there is the indication that regulators may *expect* testing as part of validation, regardless of how effective the alternative validation methods are. The conservative validation planner should plan for some testing to satisfy the regulatory issue, but should plan that testing wisely to maximize the value that the company receives from the effort. Well-planned, risk-based testing does not have to be voluminous. Yet it can be productive in finding defects that are important to the device manufacturer and may be effective in finding safety or regulatory failures.

# Nondevice Software Maintenance and Retirement Activities

## Maintenance Activities

Like medical device software, nondevice software will spend most of its life cycle in the maintenance phase. Consequently, it is important to give careful consideration to the maintenance phase activities and to plan for them as one would for any other phase of the life cycle.

The maintenance phase seems to be the forgotten phase for device software as well as for nondevice software. Why should one be concerned about maintenance phase activities? Some of the reasons were covered in the previous chapter under the topic of configuration management. Certainly, many of the configuration management activities take place during the maintenance phase of the software. Maintenance is important because during the useful life of the software problems will be discovered, unsolicited updates will be presented, user turnover will occur, the way users use the software will change, system security may be challenged, and hardware will fail, threatening the loss or corruption of data. Without adequate planning, any of these events could affect the experience with the software reducing the confidence in the results of the software. In other words, the state of validation may deteriorate over time unless the software and its validated state are maintained.

There are, of course, potential safety risks that could result from deterioration of the validated state of the software. There are also compelling business reasons why the device manufacturer should want to maintain the validated state of the software. There is the potential for loss of product, information, and time resulting from a failure of the nondevice software. The business has also made an investment in selecting and validating the software for its intended use. It would not make sense for the business to allow that investment to deteriorate over time simply because the software was not maintained.

Many of the maintenance activities for nondevice software that has been custom-developed are the same as the maintenance activities for device software. The reader is referred to Chapter 14's coverage of the maintenance activities for device software for the full complement of activities that would apply to custom-developed software. This chapter will deal primarily with the maintenance activities for nondevice software that is acquired for use (off-the-shelf).

## Release Activities

As in the discussion of maintenance activities for device software, the discussion of the maintenance phase for nondevice software will begin with some coverage of the activities associated with the release of the software for use.

### *Released Configuration*

As part of the management of the configuration of the software, the version numbers of all elements of the configuration that are released for use are recorded. Obviously, the validation activities for the software only apply to the version of software that was analyzed and/or tested on the computer system and operating system platforms upon which it was installed. In the discussion of configuration management in Chapter 18, it was pointed out that the version numbers of all other equipment and software to which the software in question interfaces is a part of the configuration of the software. A change in any part of the configuration can affect the assumptions upon which validation analysis was based, and consequently can affect the results of the experience with the software.

### *Installation and Configuration Instructions*

Some nondevice software will be deployed to hundreds or perhaps thousands of users within the organization. Instructions for the installation of the software as well as the requirements for the hardware and software operating platform should be documented to assure consistent results on the installation of the software. Some software items require some level of configuration (e.g. operating system, service pack level, database engines, etc.) upon installation. Any configuration requirements also should be specifically documented, again, to assure consistent results. If configuration files are to be supplied to the user, the location of those files should be specified. The user/installer should be provided with some mechanism for knowing which configuration file is the appropriate one to be used. (Configuration files are likely to change over time.)

Installation and configuration instructions themselves should be version controlled and tested before releasing them for use by other potential users/installers.

### *Archival Copies of Released Software*

When presented with installation instructions for a new piece of software, the installer/user will first ask where to find the specified version of software. This may seem like a trivial issue if the newly validated software is deployed to a number of users simultaneously in shiny new shrink-wrapped boxes. However, several months or years later it becomes a less trivial issue. It becomes a significant issue if the installation instructions specify the use of the validated version of software version  $n$ , but the software supplier's currently marketed version is version  $n+3$  (and version  $n$  is no longer available).

This issue has been found to be particularly serious and time-consuming for software development tools that are used to develop software for medical devices.

The development environment, which includes all of the software development tools, has to be re-created when changes or enhancements to the medical device software are needed. Sometimes this is months or years after the original release of the device software. The device manufacturer may choose to minimize the amount of change introduced to the device software by not changing the tools or the versions of the tools on subsequent releases of the device software. Finding the installation media for the full complement of software development tools included in the development environment is often a challenge. Formal archiving of these tools is a necessity, but even if they can be found it is often time-consuming and challenging to reinstall them after a significant amount of time has gone by. If nothing else, the operating system platform upon which the tools run is almost certain to change in a short period of time. Recently, we have started the practice of archiving the entire development system at the end of the project to address this significant issue. The development system includes not only the development tools, but also the hardware and operating system that is used to develop the device software.

### **Post-Release Monitoring**

With medical device software, the device manufacturer is concerned about the experience customers/users are having with the device in the field. Specifically, there is interest in any problems, confusion, misuse, or evolution of the intended use of the software. The same is true for nondevice software. The maintenance phase activities should reflect not only what the device manufacturer will do about these issues when they arise, but also how they will come to know that the issues even exist.

#### *Software Use Monitoring*

A maintenance plan should include some activities for actively seeking out and documenting the adequacy of the results of the use of the software. Also of interest are the users' satisfaction levels with the software and their opinions about what should be fixed or how its use could be improved. Gathering this information is no simple task. It is unlikely that users can be depended upon to initiate communication of this type of information. It is unlikely that they would even know to whom they should report such information. As a minimum, some facility should be provided for capturing problem reports and suggestions that do spontaneously emanate from the user population. Given the likelihood that these comments will be sparse, a better maintenance plan would include some active outreach to seek out problems the users are experiencing and ideas they have for improvement of the software.

Determining whether the intended use of the software has changed since it was validated may take a somewhat more analytical approach than simply asking users who are, most likely, unaware of what the original stated intended use was anyway. This may involve observation or analysis of results that are being produced by users who were using the software. Regardless, since the software was validated "for its intended use," it is important to understand whether the intended use has changed so that one can determine whether the software is still fit for its purpose.

Other more mechanical items should be monitored as well. There is a tendency for the more sophisticated users to install updated versions of the software whether or not they have been validated and approved by the organization. This is especially true of off-the-shelf software that may even offer automated update services through Internet connections. Changes to the process of which the software is a part, or changes to any hardware or software that interfaces with the software of interest can impact the validity of the results and thus the state of validation of the software. Consequently any active monitoring of use of the software should include monitoring of the collateral processes, software, and equipment as well.

### *Defect Management*

The details of the defect management plan may be documented in the configuration management plan, but if not, the plan for maintenance activities should cover it. Some preparation and planning should be in place to specify to whom defects are to be reported, and where those defect reports and records of their resolution will be kept. Responsibility for managing defect reports and resolutions should be assigned to a single individual. The individual will be responsible for collecting and documenting defect related data, for determining workarounds until the defects are resolved, for reporting and tracking defect resolution with the software supplier, and for deciding and planning when and how updated versions of the software will be revalidated and deployed to the users.

### **Risk Analysis and Risk Management**

Ongoing maintenance of risk analysis and the risk management plans is an important part of the maintenance phase that should take place to determine which defects require workarounds or repairs. As with device software, risk management is an activity that should take place in every phase of the life cycle of software, including the maintenance phase. Any defects or problems that are encountered during the maintenance phase should be subjected to documented risk analysis to determine how a particular defect or error could result in a risk of harm to a patient, to a bystander or operator, or to data that is critical to patient safety or regulatory process. This analysis should include consideration of the risk of *not* repairing a defect as well as the risk of introducing new software to address any defects (and potentially introducing *new* defects).

Similar analysis should be applied when considering upgrading the software for feature improvements or to add new functionality. Again, the analysis should consider the risk of introducing new defects to the software and the process it automates against the benefit to be realized from the feature improvements or new functionality.

Finally, risk analysis should consider previously unanticipated risks that are introduced by migration of the intended use of the software that results from the creativity of the users, or the introduction of new features and capabilities and upgraded software.

New risks or changed risks that are identified in this analysis and that are determined to be unacceptable will require new control measures, and a reassessment of the state of validation of the software.

## Security

Software applications that depend on user authentication, encryption, access control, or any other type of data security mechanism must be carefully monitored during the maintenance phase of the software application. These kinds of concerns are typical for electronic records/electronic signature (ERES) software. There are some specific requirements related to data security under the Part 11 regulations.

Why would data security change over time? Users of the software may leave the company, or for some other reason may no longer be authorized to access the software or its data. Users sometimes loan their passwords to coworkers introducing potential security issues. Users tend not to change their passwords as frequently as they should, and tend to use simple passwords often identical to passwords they use for other applications. All of these and other reasons result in a slow decay in the level of security of the system.

Monitoring and maintaining the state of security is really a process issue, not a software issue, unless it is found there are defects in the software related to the security mechanisms. Indirectly, however, when there is a weakness in the security policy related to the use of the software, that weakness may violate assumptions of risk control measures that include security of the data or restricted software access. The basic assumption is that the security is secure.

*Contingency planning.* Despite the best validation efforts, risk analyses, and planning, sometimes things still go wrong. It would be unwise for anyone using any software to assume that it has been validated, analyzed, and controlled well enough that there is no risk of the software failing in the future. Contingency planning should be an important part of the validation activities for the maintenance phase of the software.

This was mentioned in Chapter 16 under the topic of alternative methods of validation. A very effective alternative method (that is, an alternative to testing) of validation is to *assume that the software will fail*, and then plan effective contingencies for that event. By now, this should not sound like a radical new approach. It is nothing more than identifying risk control measures for the risks that are anticipated from software-related causes.

Regardless, some of those contingencies (or risk control measures, or mitigations if you prefer) will be implemented in the maintenance phase of the software life cycle. Some of those contingencies will require advance planning and ongoing maintenance activity. For example, contingency planning for electronic records systems might specify data recovery planning for the records data if a catastrophic software or hardware failure should damage that data. That recovery process will depend on the ongoing backup and archiving of the data to establish recovery points. If those backups are not completed as scheduled, the entire contingency plan is at risk. Verification that such ongoing maintenance activities are indeed taking place is a primary responsibility of the maintenance phase. Without

verification that such risk control measures are taking place, the risk management process would be nothing more than a meaningless, academic, paper exercise.

## Retirement of Software

Eventually, all software is retired. Perhaps it is replaced by a newer version of the same software. It may be replaced by software from another supplier. The function provided by the software may no longer be needed, or perhaps the functionality is included in new software that integrates the functions of several software items.

Regardless of the reason for retiring software, the retirement often is not a simple matter. The careful, conservative user of software should consider the impact of retiring software *before* making the decision to replace it or to retire it altogether. It seems odd, but validation is a concern even when one stops using software!

Some of the reasons software retirement becomes problematic result from:

*Reliance on the software.* Users are often not totally aware of how much they rely on software until it is taken away. Administrators may not be aware of how many users depend on the software. This really relates to *intended use*. There *should be* no surprises if the maintenance activities of the software tracked who was using the software, how the software was used, and how that use changed over the life of the software (to keep it validated for intended use). Regardless, it makes sense to assess one last time how the software was being used to accurately document the *actual* (intended) use since that is what the new software will really be trying to emulate.

*Interfaces to the software.* Retiring a software item that is replaced by another software item often results in problems with the interfaces the software has with its surroundings. Some of those interfaces included

- *Users:* Anyone who has ever upgraded their Microsoft Office applications and has felt frustrated by changes in the user interface will know this problem intimately. Changing software often changes user interfaces and can result in loss of productivity, confusion, and possibly erroneous use. Retraining or other control measures should be considered for the new software and should be in place *before* retiring the old software.
- *Equipment:* No matter how nice the new software is, there is a serious problem if it only runs on more recent computer platforms that do not support the data communication hardware and software protocols that are needed to interface to equipment and instrumentation that are part of the process. Retirement of the old software should include an accounting of all electronic interfaces so that verification of the new software can assure the successful integration with the surrounding equipment *before* retiring the old software.
- *Operating systems, supporting software and hardware:* The configuration of the old software should be well known, and the requirements for the new software should be equally obvious. What may be less obvious is that other supporting software resides on the same system with conflicting requirements

for the operating platform. Retirement of the old software should document such dependencies to facilitate a smooth transition to the new system.

- *Data:* This is perhaps the most troubling item. If data from the old software is to be preserved and used on the new software, the data is almost certain to be of a different format. A data migration task will be required to convert the format, but that will be impossible unless the data from the old system is exported or otherwise preserved *before* the system is retired and dismantled. Any documentation that existed for the old software that would have detailed the data structure will be valuable for the migration task (if the migration function is not provided by the new software). The new software as part of its validation testing will need to confirm that the data moved between systems successfully. Anything that can be provided from the old system that would assist in determining if the data transfers to the new system correctly should be collected before decommissioning of the old system. Record counts, totals, reports— anything to make a determination of the integrity of the data after transfer to the new software is valuable. This often takes some creativity to determine how a successful migration would be defined.

Too often discussions of validation of nondevice software end when the software is released for use. Since the software spends most of its life in the maintenance phase, there is often a very large window of time for software to slip from its original validated state. Detection of that slippage is not obvious and often takes a certain amount of dedication to track it and to understand how that slippage might result in previously unanticipated harm to patients, users, data integrity, or to business processes. A slip in the state of validation represents a slip of the users' confidence in the software.



# About the Author

David Vogel is the founder and president of Intertech Engineering Associates, Inc., of Westwood, Massachusetts. Founded in 1982, Intertech has served the medical device industry by providing electronics hardware and software development services. Dr. Vogel and his Intertech engineering team have developed engineering processes for product design, development, and validation that comply with the FDA Design Controls and other quality system regulations.

Dr. Vogel was selected to participate with a joint AAMI/FDA workgroup to develop a standard for critical device software validation, which was subsequently incorporated into the IEC 62304 Software Lifecycle Standard. He participated on the joint AAMI/FDA workgroup to develop a Technical Information Report (TIR32:2004) for medical device software risk management. Most recently his work on the AAMI/FDA concluded in the development of TIR36:2007 on the validation of software for regulated processes.

Dr. Vogel is a frequent lecturer at national and international tradeshows and webinars. He is a frequent contributor of articles to various industry publications on the practicalities of development and validation of medical device software in the regulated industry. He is an AAMI instructor and is currently developing a multiday workshop on practical software validation in the medical device industry. In addition to contributing content to a variety of industry trade magazines, Dr. Vogel serves on the editorial advisory board of MD&DI magazine, and is an editorial advisor to FDANews.

Dr. Vogel received a bachelor's degree in electrical engineering and computer science from the Massachusetts Institute of Technology. He earned a master's degree in biomedical engineering, a master's degree in electrical and computer engineering, and a doctorate in biomedical engineering from the University of Michigan. In 2001, the University of Michigan awarded Dr. Vogel with the Engineering Alumni Society Merit Award in Biomedical Engineering.

In 2008, MD&DI magazine honored Dr. Vogel by recognizing him as one of the "100 Notable People" in the medical device industry.



# Index

## A

- Active post-market data, 312
- Activity track life cycle model, 95–102
  - activities shrink/grow, 98
  - activity plan, 99, 100
  - defined, 96
  - dependencies, 99
  - flexibility of, 101–2
  - horizontally, 97
  - milestone definitions, 96
  - milestone tracking chart, 101, 102
  - phases, 96, 97
  - planning objectives, 99–101
  - product requirements activity track, 98
  - vertically, 97
- Ad hoc testing, 287–88
  - defined, 287
  - randomness, 288
  - speed, 287–88*See also* Testing
- After-the-fact validation, 335
- Agile development models, 68–69
- ALARP (As Low As Reasonably Predictable), 129–30, 135
- ANSI/AAMI/IEC 62304:2006 Standard
  - defined, 73
  - illustrated, 74
  - organization, 73, 74
  - risk classes, 86
- Approvals
  - defined, 48
  - expectations, 53
  - problems, 51–53
  - procedure, 49
  - regulatory basis for, 54–55
  - too many approvers and, 53
  - training for, 53
  - unclear instructions/goals for, 52
  - under duress, 52
  - value-added process, 53–54
  - in well-defined procedure, 54*See also* Signatures
- Association for the Advancement of Medical Instrumentation (AAMI), 23, 69

- TIR32:2004, 110–11
- TIR36:2007, 341, 360–61
- Assurance cases, 386–87
- Audience, this book, 10–11
- Automated process software.
  - See* Nondevice software
- Automated testing, 302–3

## B

- Black box testing. *See* System-level testing
- Boundary value testing, 279–82
  - boundaries, 280–81
  - boundary conditions, 280, 281–82
  - equivalence class partitioning and, 279–80

## C

- Calculations and accuracy testing, 282–86
  - defined, 282
  - floating point addition, 282–83
  - floating point calculations, 285–86
  - floating subtraction, 284–85*See also* Testing
- Captured defect testing, 288–89
  - defined, 288–89
  - procedures, 289*See also* Testing
- Center for Devices and Radiological Health (CDRH), 16–17
  - defined, 16
  - Office of Compliance, 16
  - Office of Device Evaluation (ODE), 16
  - Office of Science and Engineering Laboratories (OSEL), 250
  - Office of Science and Technology (OST), 17
- Change control, 158–59
  - board (CCB), 159
  - process, 159
- Change management
  - defect management as part of, 162
  - procedure, 161
- Coding standards/guidelines, 248
- Combined development and validation
  - waterfall life cycle model, 91–93
- defined, 91

- Combined development and validation  
 waterfall life cycle model (continued)  
 illustrated, 92  
 parallel activities and, 92–93  
 validation activities, 91
- Command dictionaries, 269
- Commercial off-the-shelf software (COTS), 334
- Communication links, 239–40
- Compound requirements, 219–20, 224–25
- Concept phase, 193–94  
 activities, 193–205  
 defined, 193  
 OTS software decision, 198–99  
 regulatory background, 194–95  
 validation activities during, 196–98  
*See also* System requirements specification (SyRS)
- Configurable software, 334
- Configuration items (CIs), 156  
 externally developed, 159–60  
 naming, 157  
 storage, 157–58
- Configuration management (CM), 153–60  
 defect management relationship, 161–65  
 defined, 153  
 reasons for, 154–55  
 regulatory background, 153–54
- Configuration management (nondevice software), 387–91  
 acquired-for-use software, 390  
 activities, 391  
 compatibility, 388–89  
 importance, 388–89  
 intended use, 388  
 intended use environment, 388  
 intended users, 388  
 issues, 387  
 planning, 389–91
- Configuration management plans (CMPs), 153  
 change control, 158–59  
 configuration items (CIs) itemization, 156  
 configuration items (CIs) naming, 157  
 contents, 155–60  
 externally controlled items coordination, 159  
 externally developed configuration items control, 159–60  
 external software tools itemization, 156–57  
 nondevice software, 389–91  
 as plans, 156  
 storage details, 157–58
- version control, 157
- Consistency, software and documentation, 84
- Contingency planning, 405–6
- Customized/hybrid-partially off-the-shelf (OTS) software, 334
- Cyclomatic complexity (CC)  
 calculations, 264  
 defined, 264  
 extended measure, 267  
 McCabe metrics, 265–66
- D**
- Defect management, 160–66  
 analysis, 163  
 classification, 162–63  
 closure, 164  
 configuration management relationship, 161–65  
 evaluation and prioritization, 163  
 identification, 162  
 implementation, 164  
 importance, 161  
 issues, 160–61  
 nondevice software, 404  
 as part of change management, 162  
 planning for, 165–66  
 problem reports, 160  
 procedures, 161, 162–64  
 recommendation, 163–64  
 regulatory background, 161  
 state diagram representation, 166  
 verification, 164
- Design activities, 233–47  
 communication links, 239–40  
 design reviews, 239  
 evaluations, 239  
 introduction to, 233–34  
 regulatory background, 234–36  
 risk management, 246–47  
 software design specification (SDS), 236–39  
 traceability analysis, 240–46  
 trace schema, 245  
 validation tasks related to, 236–47
- Design and development planning, 143–44  
 describe or reference, 144  
 establish and maintain, 143–44  
 interfaces, 144  
 responsibility, 144  
 updated and approved, 144  
*See also* Planning
- Design Control Guidance for Medical Device Manufacturers, 33, 108

Design Controls, 17–18, 20–22

  Design Validation, 21–22  
  length of, 21  
  subsections, 21

Design controls

  review role and, 169  
  validation and, 84–85  
  verification and, 84–85

Design History File (DHF), 19

Design requirements

  examples, 242  
  traceability, 240–41

Design reviews, 170, 171, 239

Design validation

  activities, 43  
  defined, 32  
  regulation for, 33

Design verification, 32

Detectability, 128–29

Device communications testing, 269–72

  command dictionary, 269  
  configuration, 270  
  debuggers for, 270  
  defined, 269  
  in-circuit emulators for, 270  
  long-run tests, 271  
  regression testing, 271  
  specifications, 269  
  stress tests, 272  
  test setup block diagram, 271  
  timing analysis, 271

*See also* Testing

Device master record (DMR), 371

Documentation

  of compiler outputs, 249–50  
  consistency, 84  
  procedure and environment, 310–11  
  software version agreement, 310  
  testing, 289

Document controls, 55

Document thrashing, 238

Do it yourself (DIY) software, 343

Do-it-yourself (DIY) validation, 347–49

  building blocks, 348–49

  ingredients, 348

  for nonsoftware engineers, 347–49

## E

Electronic records and electronic signatures

  (ERES), 34, 328

  software validation, 328–29

  system users, 328

Embedded software, 334

Environment

  intended use, 202–3, 388  
  legacy, 203  
  platform, 204  
  software use, 367–68  
  standards, 203

Equivalence class partitioning, 277, 279–80

Equivalence class testing, 276–79

  classes, 277–78  
  concurrent classes division, 278  
  defined, 277  
  prioritization requirements, 278  
  unanticipated situations, 279

*See also* Testing

Error guess testing, 286–87

  defined, 286  
  effectiveness, 287  
  examples, 286  
  process, 286–87

*See also* Testing

Extended cyclomatic complexity measure, 267

Extreme programming (XP), 68

## F

Failure modes and effects analysis (FMEA),  
  128, 139

Failures

  detectable, 129  
  nondevice software, 377  
  number of users affecting probability, 338  
  qualitative probability of, 122–29  
  random, 109  
  software, qualitative probability, 125–26  
  systematic, 109

Fault tree analysis (FTA), 139

FDA

  beginnings, 14  
  Center for Devices and Radiological Health  
    (CDRH), 16–17  
  Design Controls, 17–18, 20–22  
  guidance documents, 23  
  laws and regulations as tools, 18  
  Medical Device Amendments, 19  
  Medical Device Reports (MDRs), 20  
  mission statement, 16  
  1906 through 1990, 13–16  
  organizational chart, 17–18  
  premarket approvals (PMAs), 14, 20  
  product submission approval, 20  
  quality system approval, 20  
  reactive nature, 13–14

FDA (continued)

- responsibility of, 13
- safety, efficacy, security assurance, 17–20
- software validation regulations, 27–35
- staff, 16
- standards database, 23
- Therac-25 and, 15
- today (2010), 16–17

*See also* General Principles of Software Validation (GPSV); Quality System Regulations (QSRs)

Food and Drugs Act of 1906, 14

Food Drug and Cosmetic Act (FDCA), 14, 19

Formal testing, 298–300

- defined, 298–99
- progression, 299
- results examination, 299
- start of, 299

*See also* Testing

## G

General principles of Software Validation (GPSV), 31, 35, 43

- activities, 76, 77
- automated processes, 326–28
- configuration management planning, 153
- defect management, 161
- design activities, 235
- electronic records and electronic signatures, 328–30
- integration-level testing, 268
- intended uses, 82, 363
- IQ/OQ/PQ, 397
- nondevice software, 326–30
- output verification, 357
- reviews, 169
- software life cycle, 60
- software requirements, 208–10
- software validation plan, 147–48
- software verification, 82–83
- system-level testing, 272–73
- testing, 253–54, 260
- traceability, 80, 178–79

Global Harmonization Task Force (GHTF), 25

Goals, this book, 9–10

*Good Automated Manufacturing Practice* guide, 341–42

Good manufacturing practices (GMPs), 15

Governing documents, 7

Guidance documents, 23

## H

Halstead length measure, 267

Harms

- defined, 115
- nondevice software, 376–77
- RCM relationship, 132
- software failure relationship, 377

Hazard analysis and critical control point (HACCP), 139

Hazard-harm analysis, 121

Hazard operability study (HAZOP), 139

Hazardous situations

- defined, 115
- RCM relationship, 132

Hazards

- analysis, probabilities distribution in, 119
- analysis for, 117
- defined, 115
- identification of, 117–18
- progression to risk, 116
- RCM relationship, 132

Head-in-the-sand traceability, 243

High-level architecture (HLA), 238

## I

IEC/TR 80002-1, 112

IEEE

- Standard for Developing Software Life Cycle Processes, 72–73
- Standard for Software Configuration Management Plans, 155, 156
- Standard for Software Reviews, 172

Implementation activities, 247–50

- coding standards and guidelines, 248
- documentation of compiler outputs, 249–50
- software components reuse, 248–49
- source code creation, 247
- static analysis, 250
- validation tasks related to, 247–50

Impromptu testing. *See* Ad hoc testing

Independence, test, 296–97

Inferred traceability, 185–86

Informal testing, 297–98

Installation qualification (IQ), 397–99

Integration-level testing, 267–72

- defined, 260
- device communications, 269–72
- GPSV, 268
- priority order, 268
- project uniqueness, 269

*See also* Testing

- Intended use, 364–69  
defined, 82  
determining, 366–69  
environment, 202–3, 388  
fulfilling, 369–74  
GPSV, 363  
necessity to state, 364–69  
nondevice software, 336–37, 365, 388  
reasons for stating, 364–65  
risk analysis specification, 117  
software function, 367  
software use environment and, 367–68  
statement contents, 365–66  
validation for, 364–65  
when to be used, 368–69
- Intended use requirements, 369–74  
for acquired software, 370  
for custom-developed software, 369–70  
example, 372–74  
information content of, 370–72  
as quantitative measures, 372  
support, 371  
types of, 372  
verification, 371
- Intended users  
nondevice software, 337–40, 388  
probability of error and, 338  
in statement of intended use, 367  
system requirements specification (SyRS),  
202
- ISO-13485, 24
- ISO-14971, 108–10  
application to medical device software, 112  
applying to nondevice software, 375–76  
defined, 108  
random failures, 109  
systematic failures, 109
- L**
- Life cycles  
activity track model, 95–102  
advantages, 89  
approach to software validation, 89–103  
combined development and validation,  
91–93  
industry standards and, 102–3  
model objections, 95–96  
model selection, 103  
out-of-the-box nondevice software, 352  
parallel, 94, 95  
planning of validation, 350–52  
processes, 111–12
- programmable software, 353  
validation and, 90–91  
validation model, 93–95  
*See also* Software development life cycle  
(SDLC)
- Lines of code (LOC), 267
- Long-run tests, 271–72
- Low-level design detail (LLD), 237–38
- M**
- Maintenance, 305–21  
activities, responsibility for, 307  
activities model, 308–12  
analysis, 317–18  
development phases relationship, 306  
difficulty, 307  
introduction to, 305–7  
lengthening of, 307  
nondevice software, 401–6  
phase, 305–21  
post-market data analysis, 315–18  
post-market data collection, 312–15  
risk, 307  
software release activities, 309–12, 321  
test plan for release, 320
- Maintenance software development life cycles,  
318–21  
legacy software, documentation, test  
procedures, 318–19  
regulatory changes, 319–20  
software development and validation  
activities, 320–21
- Management awareness, 145
- Management responsibility, 38
- Management reviews, 38, 171–72
- Management with executive responsibility, 37
- Mars Climate Observer (MCO), 239–40
- Master copies, 311–12
- McCabe cyclomatic complexity metric, 263–66  
calculating, 263  
defined, 264  
path coverage and, 263–66
- Medical Device Amendments (MDA), 15, 19
- Medical Device Directives (MDDs), 24
- Medical device industry (MDI), 13
- Medical Device Reports (MDRs)  
defined, 20  
reduction in, 35
- Medical devices  
risk management application to, 108–10  
sold outside U.S., 24–25

Medical device software  
 application of ISO 14971 to, 112  
 development as tug-of-war, 11  
 risk management, 8, 106–41  
 validation of, 8  
**Metrics**, 62  
**Milestone tracking chart**, 101, 102  
**Modified waterfall SDLC model**, 63, 64  
**Module testing.** *See* Unit-level testing

**N**

**Negative requirements**, avoiding, 226  
**Nondevice software**  
 backup, recovery, and contingency planning, 358–59  
 complexity, 336  
 confidence in source, 337  
 configurable, 334  
 configuration management, 387–91  
 configuration management plans (CMPs), 389–91  
 contingency planning, 358–59, 405–6  
 COTS, 334  
 covered by regulations, 330–32  
 customized/hybrid-partially OTS, 334  
 defect management, 404  
 DIY, 343  
 DIY validation, 347–49  
 embedded, 334  
 failure, 377  
 industry guidance, 340–42  
 influences affecting validation, 335–36  
 intended use, 336–37, 365  
 intended users, 337–40  
 known issue analysis, 355  
 level of control, 332–34  
 life cycle planning of validation, 350–52  
 maintenance activities, 401–6  
 maintenance of risk analysis, 404–5  
 OTS, 334  
 output verification, 357–58  
 planning validation for, 345–61  
 post-release monitoring, 403–4  
 proactive validation activities, 359  
 product selection, 354  
 programmable, 334  
 regulatory background, 326–30  
 release activities, 402–3  
 retirement activities, 406–7  
 risk, 336  
 risk management, 375–87  
 safety in numbers, 355–56

security maintenance, 405–6  
 security measures, 359  
 size, 336  
 software use monitoring, 403–4  
 source of, 334  
 supplier selection, 354–55  
 third-party validation, 356–57  
 training, 360  
 type of, 334  
 validation activities factors, 332–36  
 validation background, 325–43  
 validation moderators, 347  
 validation plan, 360–61  
 validation spectrum, 349–50  
 validation toolbox, 352–60  
 waterfall model, 351  
 who should be validating, 342–43  
**Nondevice software release activities**  
 archival copies, 402–3  
 installation and configuration instructions, 402  
 released configuration, 402  
**Nondevice testing**, 393–400  
 acquired software, 396–97  
 installation qualification (IQ), 397–99  
 operational qualification (OQ), 397–99  
 performance qualification (PQ), 397–99  
 reasons for/against, 393–94  
 regulatory realities, 395–96  
 as risk control measure, 395  
 summary, 400  
 validation of Part 11 regulated software, 399

**O**

**Objective evidence**, 80–81  
**Occam's razor**, 138–40  
**Office of Compliance**, 16  
**Office of Device Evaluation (ODE)**, 16  
**Office of Science and Engineering Laboratories (OSEL)**, 250  
**Office of Science and Technology (OST)**, 17  
**Off-the-shelf (OTS) software**, 87  
 commercial (COTS), 334  
 configurable, 334  
 customized/hybrid-partially, 334  
 decision, 198–99  
 nondevice software, life cycle model, 352  
 programmable, 334  
 testing, 198  
 use considerations, 198–99  
**Operational qualification (OQ)**, 397–99

- Organization, SDLC, 62  
Organization, this book, 8–9, 73–74  
Organizational responsibility  
  analytical view, 45–49  
  approvals and signatures regulation, 54–55  
  approval/signature process, 53–54  
  management with executive responsibility, 37  
  quality system failure, 49–53  
  regulatory basis of, 37–39  
  for software validation, 37–55  
Output verification, 357  
  GPSV, 357  
  use of, 357, 358  
  validity and application of, 358  
Overall residual risk evaluation, 134–40  
  combined risks, 135  
  Occam's razor, 138–40  
  proper presentation, 136  
  relative comparisons, 137–38  
  rule-based approach, 136–37  
  safety assurance cases, 138  
*See also* Risk control
- P**
- Parallel life cycles, 94, 95  
Passive post-market data, 312  
Path coverage  
  McCabe cyclomatic complexity metric and, 263–66  
  unit-level testing and, 263  
Peer reviews, 171  
Performance qualification (PQ), 397–99  
Planning, 143–52  
  configuration management, 153  
  configuration management (nondevice software), 389–91  
  for defect management, 165–66  
  design and development, 143–44  
  importance, 144–45  
  management awareness, 145  
  nondevice software validation, 345–61  
  schedules, 145
- Plans  
  approach, 150  
  configuration management (CMP), 155–60  
  defined, 42  
  description of tasks, 151  
  evolving, 152  
  generic outline, 149  
  issue management, 150–51  
  life-cycle description, 150  
  references, 149–50  
  relevant quality factors, 150  
  required, 145–47  
  review strategy, 150  
  risks and assumptions, 152  
  roles, responsibilities, and resources (3 Rs), 152  
  scope, 149  
  structure and content, 147–48  
  team following, 101  
  traceability strategy, 151  
  validation structure, 146
- Policies, 41
- Post-market data  
  active, 312  
  analysis, 315–18  
  change requirements, 315–16  
  changes to external interface, 314  
  collection of, 312–15  
  competitive information, 314  
  complaints, 313  
  new features/functions requests, 313  
  new opportunities, 314  
  passive, 312  
  performance improvements, 314  
  process and planning, 313  
  regulatory change, 314  
  revised risk management documents, 316  
  sources, 313–15  
  supporting documentation, 316–17  
  technology changes, 314  
  user groups/web sites, 314
- Preliminary hazard analysis (PHA), 139
- Premarket notifications (PMNs), 19
- Probability  
  defined, 115  
  distribution in hazard analysis, 119  
  estimate scenarios, 381  
  ignoring, 123  
  nondevice software, 378–81  
  qualitative, 123–29  
  quantitative analysis, 118–29
- Problem reports, 160
- Procedures, 5  
  captured defect, 289  
  defect management, 161, 162–64  
  defined, 42  
  quality system, 43–45  
  review, 175  
  review and approval, 49, 54  
  test, 257, 292–95

- Processes, 5  
 defined, 41  
 life cycle, 111–12  
 quality system, 42–43  
 risk management, 112–13  
 software development, 102–3  
 validation, 32
- Product Requirements Definition (PRD), 50–51
- Programmable software, 334
- Prospective validation, 335
- Q**
- Qualitative probability analysis, 122–29  
 detectability, 128–29  
 levels of probability, 124  
 probability estimates, 123  
 risk evaluation tables, 123–24  
 severity, 126–28  
 software failure, 125–26
- Qualitative risk acceptability criteria, 386
- Quality assurance activities, 43
- Quality planning, 38
- Quality policy, 38
- Quality System Regulations (QSRs), 17, 20–22  
 authoring of, 31  
 definitions section, 32  
 design activities, 234  
 executive responsibility, 37  
 length of, 21  
 management responsibility, 38  
 nondevice software, 330  
 perceptions, 21  
 preamble, 21, 32  
 reviews, 168  
 software development life cycles and, 70–73  
 software validation requirement, 35  
 stages of device design development, 60
- Quality systems  
 assurance of compliance, 41  
 components automated by software, 331  
 consistency of quality, 41  
 corporate memory, 41  
 defined, 39  
 flexibility, 41  
 life cycle planning in, 72  
 model for, 39–45  
 procedures, 38, 42, 43–45  
 processes, 41, 42–43  
 process model illustration, 40  
 review, revise, and approve (RR&A), 39–40, 51–53  
 roles, responsibilities, and goals, 40–41  
 structure of, 41–42  
 swim lane diagram, 42  
 as system, 39  
 terminology, 41–42  
 tracing unit-level testing through, 43–45  
 what could go wrong with, 49–53
- Quantitative probability analysis, 117–22  
 cross-check, 120  
 estimates, 122  
 in risk analysis, 118–22
- Quantitative risk acceptability criteria, 384–85
- R**
- Random failures, 109
- Randomness, test, 295–96
- RASCI charts  
 defined, 45  
 illustrated, 46  
 in project plan, 47  
 in reviews, 172  
 roles definition, 46
- Readability requirements, 227–28
- Regression testing, 271, 300–302  
 defined, 300  
 guidelines, 300–302  
 test result matrix, 300, 301  
 test selection, 300  
*See also Testing*
- Regulations  
 design validation, 33  
 objective, 22  
 relationship with job completion, 22–23  
 risk in, 107–8  
 software validation, 27–35
- Regulatory background  
 concept phase, 194–95  
 configuration management (CM), 153–54  
 design activities, 234–36  
 medical device industry, 13–25  
 nondevice software, 326–30  
 reviews, 167–68  
 risk management, 107–8  
 software requirements, 208–10  
 testing, 253–55  
 traceability, 178–81
- Relative comparisons, 137–38
- Requirements. *See Design requirements;*  
 Intended use requirements;  
 Software requirements
- Responsibility, creation and maintenance of documents, 47

- Retirement of software, 406–7  
Reusing software, 248–49  
Review, revise, and approve (RR&A), 39–40  
    defined, 39  
    designing, 51–53  
    regulatory requirements for, 54  
    requirements, 40  
    *See also* Approvals; Signatures  
Reviewers, 173  
Reviews, 167–77  
    attitude towards, 176  
    attributes, 169  
    conducting, 173–77  
    design, 170, 171, 239  
    findings, dealing with, 176–77  
    findings, difficulty scale, 173  
    importance of, 168–70  
    management, 171–72  
    organizational relationship with, 169  
    participants in, 172–73  
    peer, 171  
    preparation, 175  
    procedures, 175  
    process definition, 174–75  
    reader, 175–76  
    regulatory background, 167–68  
    requirements, 168  
    role in design control process, 169  
    session length, 175  
    technical, 171  
    usability, 169  
    when to start, 174  
Risk  
    acceptability, 129  
    combined, 135  
    defined, 115  
    estimation of, 118  
    evaluation, 129–30  
    hazard progression to, 116  
    IEC 62304 standard and, 111–12  
    maintenance phase, 307  
    nondevice software, 378–81  
    in regulations and guidance documents, 107–8  
    regulatory, mapping to safety, 379  
    validation relationship, 107  
Risk acceptability, 383–87  
    criteria definition, 383–84  
    qualitative criteria, 386  
    quantitative criteria, 384–85  
    safety assurance case method, 386–87  
Risk analysis, 117–22  
    defined, 117  
    as design activity, 246–47  
    hazard analysis approach, 125  
    hazards identification, 117–18  
    intended use specification, 117  
    objective of, 127  
    quantitative probability analysis, 118–22  
    risk estimation, 118  
    safety characteristics identification, 117  
Risk-based validation, 106  
Risk classes, 86  
Risk control, 130–40  
    activities, 113  
    activity, 133  
    measures, 113  
    overall residual risk evaluation, 134–40  
    relationships analysis, 133  
Risk control measures (RCMs), 130–40  
    defined, 130  
    hazards, hazardous situations, harm  
        relationship, 132  
    methods, 133  
    testing, 134  
    types of, 131–32  
Risk management, 105–41  
    AAMI TIR32:2004, 110–11  
    activities, 117–29  
    application to medical devices, 108–10  
    components relationship, 109  
    concepts, 113, 115–17  
    definitions, 113, 115–17  
    as design activity, 246–47  
    design input requirements, 108  
    file, 115  
    importance, 106  
    language of, 113–14  
    of medical device software, 8  
    medical device software, 110–11  
    outputs, 114–15  
    plan, 114–15  
    process, 108, 112–13  
    qualitative probability analysis, 122–29  
    reduced test activity as byproduct, 106  
    revised documents, 316  
    risk analysis, 117–22  
    role in requirements development, 214–15  
Risk management (nondevice software), 375–87  
    applying 14971 process to, 375–76  
    detectability, 387  
    harm, 376–77  
    process control, 383

- Risk management (nondevice software)  
 (continued)  
 risk, severity, probability, 378–81  
 risk acceptability, 383–87  
 validation and, 382–83
- Rule-based system, 136–37
- S**
- Safe Medical Devices Act, 15  
 Safety assurance cases, 138  
 Safety tracing, 178  
 Sashimi modified waterfall model, 63–66  
 defined, 63  
 illustrated, 65  
 left to be determined (TBD), 64–66  
 management, 64  
 requirements evolution under, 65  
 test phase, 66  
*See also* Software development life cycle (SDLC)
- Schedules, 145  
 Security  
 maintenance (nondevice software), 405–6  
 nondevice software, 359  
 Session-based testing (SBT), 289  
 Severity, 126–28  
 defined, 115  
 levels of, 127  
 nondevice software, 378–81  
 regulatory risk, 380  
*See also* Qualitative probability analysis
- Signatures  
 delayed, 52  
 in design control process, 55  
 importance of, 49  
 problems, 51–53  
 regulatory basis for, 54–55  
 sniping, 53  
 under duress, 52  
 value-added process, 53–54  
*See also* Approvals
- Soft requirements, 222  
 Software  
 coding, 248  
 components resuse, 248–49  
 consistency, 84  
 documentation version agreement, 310  
 harm to users/patients, 33  
 label, package, delivery, 312  
 master copies, 311–12  
 OTS, 87  
 retirement of, 406–7
- version control system (VCS), 311  
*See also* Medical device software; Nondevice software
- Software design description (SDD), 154, 236  
 Software design specification (SDS), 236–39  
 contents, 237  
 defined, 235, 236  
 elements, 235–36  
 external document references, 236  
 high-level architecture (HLA), 238  
 low-level design detail (LLD), 237–38  
 need for, 237  
 sections, tracing to, 246
- Software development  
 activities, 43  
 process, 102–3
- Software development life cycle (SDLC), 57–74  
 activities, 60  
 agile development models, 68–69  
 benefits, 61  
 characteristics, 57  
 defined, 57–59  
 hierarchy illustration, 59  
 high-level, 58  
 maintenance, 318–21  
 metrics, 62  
 model importance, 61–62  
 models, 57, 59, 62–69  
 model selection, 69–70  
 modified waterfall model, 63, 64  
 organization, 62  
 quality system and, 70–73  
 sashimi modified waterfall model, 63–66  
 several, advantages/disadvantages, 71  
 software validation and, 60  
 spiral model, 66–68  
 waterfall model, 63
- Software of unknown provenance (SOUP), 198  
 Software release activities, 309–12, 321  
 completion of testing verification, 309–10  
 label, package, delivery, 312  
 master copies, 311–12  
 procedure/environment build  
 documentation, 310–11  
 software/documentation version agreement, 310  
*See also* Maintenance
- Software requirements  
 accuracy, 224–31  
 activities, 207–31  
 anatomy, 219–23  
 as basis for system-level testing, 213

- clarity, 224–31  
for communication, 212  
for complex logic, 228  
components, 219  
compound, 219–20  
compound, avoiding, 224–25  
context, 218  
in controlling feature creep, 212  
design debate, 218  
as design input, 211–12  
examples, 242  
exceptions, avoiding, 225  
good, 223–31  
growing reliance on, 211  
hierarchical organization of, 220  
for high-level debugging, 214  
importance of, 210–14  
introduction to, 208  
maximum visibility in creation, 215  
negative, avoiding, 226  
phase, 207–31  
readability, 227–28  
regulatory background, 208–10  
risk management role in development, 214–15  
soft, 222  
summary, 231  
synonyms, avoiding, 225–26  
template, 228–30  
testability, 227  
traceability, 180, 181, 226–27  
who should write, 215–17  
as working document, 212–13  
writing challenge, 215  
*See also* Software requirements specification (SRS)
- Software requirements specification (SRS), 154–55  
defined, 208  
requirements summary in, 217  
reviewing, 218  
sections, tracing, 246  
soft language, 222  
validation, 208  
verification, 208  
Software unit test tools, 261–62  
Software validation  
life cycle approach, 89–103  
organizational considerations, 37–55  
reasons for, 34–35  
regulatory guidance, 79–81  
requirement by law, 35  
risk management importance, 106–7  
systematic program, 35  
testing, 34, 43  
*See also* Nondevice software  
Software validation regulations, 27–35  
confidence building, 30–31  
hierarchy, 18  
reason for, 27–28  
Therac 25 incident and, 28–30  
Software verification, 43  
GPSV, 82–83  
regulatory guidance, 82–84  
in software life cycle, 60  
tasks, 83  
Spaghetti traceability, 244  
Spiral SDLC model, 66–68  
defined, 66  
illustrated, 67  
quadrants, 67–68  
use of, 68  
versions, 67  
*See also* Software development life cycle (SDLC)  
Standard operating procedures (SOPs), 10  
Standards  
coding, 248  
compliance with, 23  
database, 23  
industry, life cycles and, 102–3  
traceability to, 183  
Static analysis tools, 250  
Stress tests, 272  
Structural testing. *See* Unit-level testing  
Supporting activities  
configuration management, 153–60  
defect management, 160–66  
defined, 105  
planning, 143–52  
reviews, 167–77  
risk management, 106–41  
traceability, 177–89  
types of, 105  
Systematic failures, 109  
System-level testing, 272–75  
defined, 260  
design detail and, 260  
elements, 272  
GPSV, 272–73  
subcategories, 272  
validation, 274–75  
verification, 274–75  
*See also* Testing

- System requirements specification (SyRS),  
     194–95, 200  
     activities, 195  
     as basis for validation testing, 196  
     contents, 201–5  
     control measures, 215  
     development of, 200  
     intended audience, 200–201  
     intended use, 201–2  
     intended use environment, 202–3  
     intended users, 202  
     legacy environment, 203  
     need for, 195–96  
     organization, 204  
     platform environment, 204  
     responsibility, 196–97  
     standards environment, 203  
     *See also* Concept phase
- Systems engineering functions, 43
- T**
- Technical definitions, 5  
 Technical documents, 8  
 Technical evaluations, 171  
*Technical Information Report AAMI TIR32:2004 on Medical Device Software Risk Management*, 8  
 Technical information reports (TIRs), 23, 69  
 Templates requirements, 228–30  
 Terminology  
     boring documents and, 7–8  
     correct versus consistent, 6–7  
     definition deliberation, 5  
     developer use of, 4–5  
     as foundation, 5–6  
     function of, 7–8  
     with governing documents, 7  
     in hierarchy tree, 7  
     quality system structure, 41–42  
     well-ordered, 6  
 Testability requirements, 227  
 Test cases, 291–92  
     defined, 291  
     second-order advantages, 291  
     table example, 292  
     test designs/procedures relationship, 292  
 Test designs, 290–91  
 Test-driven development (TDD), 68  
 Testing, 253–303  
     activities, 253–303  
     actual results, 257  
     ad hoc, 287–88  
     assessment, 257  
     automated, 302–3  
     boundary value, 279–82  
     calculations and accuracy, 282–86  
     captured defect, 288–89  
     confidence and, 255  
     device communications, 269–72  
     documentation, 289  
     equivalence class, 276–79  
     error guess, 286–87  
     exercising versus, 257–58  
     expected behavior and, 255  
     expected results, 256–57  
     formal, 298–300  
     function, 76  
     informal, 297–98  
     integration-level, 260, 267–72  
     introduction to, 253  
     levels of, 260–75  
     levels representation, 161  
     long-run tests, 271–72  
     methods, 275–90  
     nondevice, 393–400  
     OTS software, 198  
     procedure step, 257  
     psychology of, 258–60  
     reasons for, 255–56  
     regression, 271, 300–302  
     regulatory background, 253–55  
     as risk control measure, 134  
     roots of, 262–63  
     session-based (SBT), 289  
     software, defining, 256–58  
     specified conditions, 256  
     stress tests, 272  
     summary, 303  
     system-level, 260, 272–75  
     terminology, 274  
     unit-level, 260, 261–66  
     validation, 274–75  
     validation, SyRS as basis, 196  
     validation relationship, 76–78  
     verification, 274–75  
 Test management, 295–302  
     formal testing, 298–300  
     independence, 296–97  
     informal testing, 297–98  
     randomness, 295–96  
     role of, 259  
 Test procedures, 257, 292–95  
     advantages, 292–93  
     assessment decision, 294–95

- column formatting, 293–94  
disadvantages, 293  
results assessment, 294  
test designs/cases relationships, 292
- Test teams, training, 259
- Therac 25  
defined, 15  
detail importance, 30  
factors leading to disaster, 28–30  
as FDA eye-opener, 15
- Third-party validation, 356–57
- Timing analysis, 271
- Traceability, 177–89  
beyond regulatory guidance, 182–85  
detailed design to source code, 180–81  
detailed design to unit level tests, 181  
GPSV, 79–80  
head-in-the-sand, 243  
high level design to integration level tests, 181  
how it's done, 185–89  
inferred, 185–86  
metrics, 177  
origins of requirements, 177  
overdoing, 189  
reasons for, 177–78  
regulatory background, 178–81  
requirements, 180, 181, 226–27  
resolution of, 246  
risk analysis to software requirements, 180  
safety, 178  
software requirements to high-level/detailed design, 180  
software requirements to system level  
software tests, 181  
source code to unit level tests, 181  
spaghetti, 244  
to standards, 183  
to system-level requirements, 79  
system requirements to/from software requirements, 180  
top-down scheme, 182  
trace mapping, 188–89  
trace tools, 185–87  
unit level tests to risk analysis results, 181
- Traceability analysis, 178  
as design activity, 240–46  
facilitation, 169  
software requirements to design description, 241
- Traces  
analysis, 187
- back, 186  
creation, 187  
low-resolution, 186  
many-to-many mapping, 189  
many-to-one mapping, 188  
mapping, 188–89  
matrix, 186  
morphologies, 188–89  
one-to-many mapping, 188  
one-to-one mapping, 188  
tools for, 185–87  
unique numbering of requirements, 186–87
- Training  
for approvals, 53  
nondevice software, 360  
test teams, 259
- U**
- Unit-level testing, 261–67  
defined, 260  
objectives, 262  
path coverage and, 263  
prioritization, 267  
software unit test tools, 261–62  
unit under test (UUT), 261, 262  
as verification test activity, 262  
*See also Testing*
- Unit under test (UUT), 261, 262
- User needs, 82
- Utah Medical Products case, 395–96
- V**
- Validation  
activities, concept phase, 197–98  
activities, placing in validation zones, 78  
after-the-fact, 335  
commensurate with complexity and risk, 85–87  
defined, 32  
design, 32, 33  
design controls and, 84–85  
design-related tasks, 236–37  
development life cycle and, 3–4  
DIY, 343, 347–49  
during concept phase, 196–98  
equality, 87  
evolution of, 3–4  
good engineering practices and, 78  
as group activity, 258  
implementation-related tasks, 247–50  
for intended use, 364–65

- Validation (continued)  
language building for, 4–8  
life cycles and, 90–91  
of medical device software, 8  
miscommunication problem, 4  
misconceptions, 75–76  
moderators, 347  
nondevice software, 325–61  
plan structure, 146  
process, 32  
prospective, 335  
risk-based, 106  
risk relationship, 107  
SRS, 208  
testing, 274–75  
third-party, 356–57  
umbrella, 77  
verification and testing relationship, 76–78  
V model for, 90  
zones, 78  
*See also* Software validation; software validation regulations
- Validation life cycle model, 93–95  
defined, 93–94  
design phase, 94
- parallel life cycles, 94, 95  
reviewed and approved test procedures, 94
- Verification  
activities, 76  
design controls and, 84–85  
intended use requirements, 371  
misconceptions, 75–76  
nondevice software output, 357–58  
software/documentation version agreement, 310  
SRS, 208  
testing, 274–75  
testing completion, 309–10  
validation relationship, 76–78  
*See also* Software verification
- Version control, 154, 157
- Version control system (VCS) software, 311
- V model, 90
- W**
- Waterfall SDLC model, 63  
defined, 63  
illustrated, 63, 91  
for nondevice software, 351
- White box testing. *See* Unit-level testing