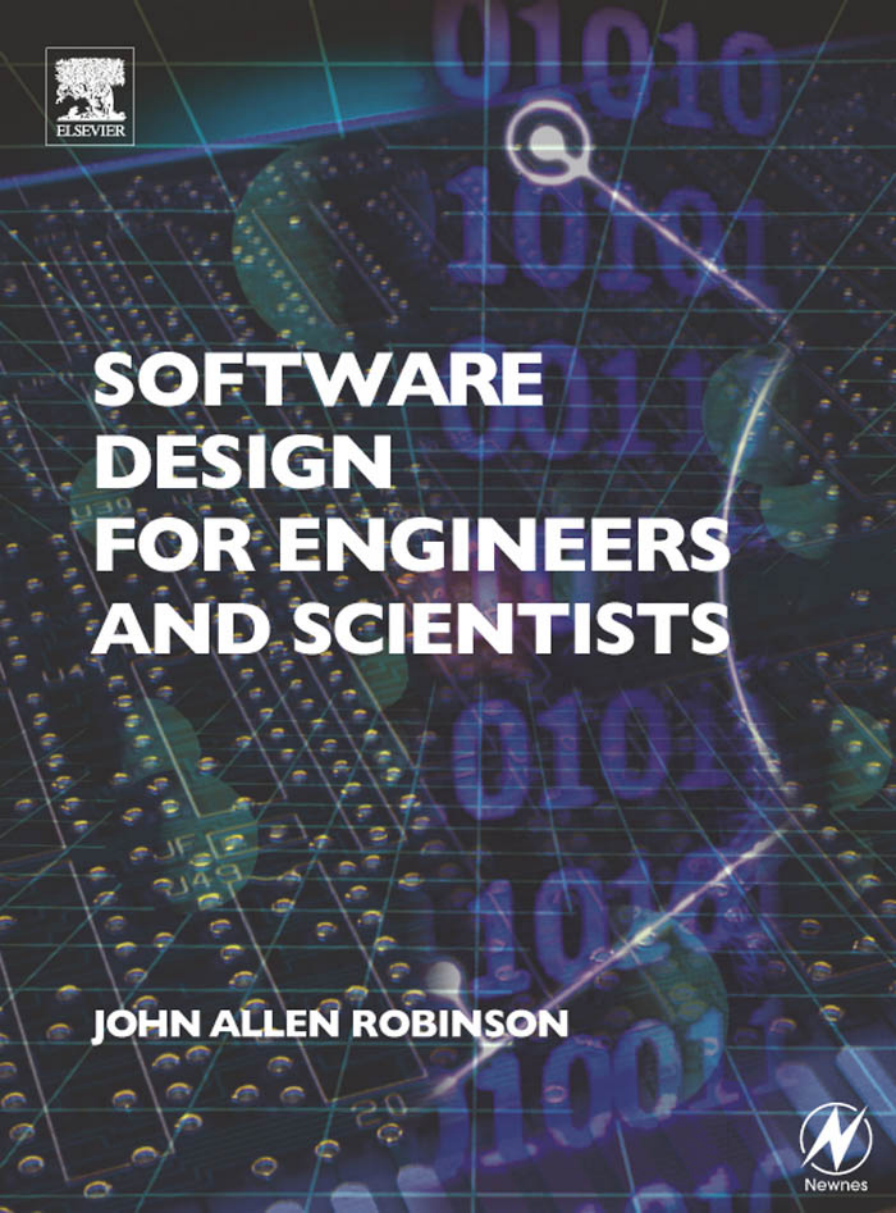
ELSEVIER
SOFTWARE DESIGN FOR ENGINEERS AND SCIENTISTS
JOHN ALLEN ROBINSON
Newnes

Software Design for Engineers and Scientists

This page intentionally left blank

Software Design for Engineers and Scientists

John A. Robinson
University of York

Newnes

AMSTERDAM • BOSTON • HEIDELBERG • LONDON • NEW YORK • OXFORD
PARIS • SAN DIEGO • SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Newnes
An imprint of Elsevier
Linacre House, Jordan Hill, Oxford OX2 8DP
30 Corporate Drive, Burlington, MA 01803

First published 2004

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

ISBN 0 7506 6080 5

Typeset by Charon Tec Pvt. Ltd, Chennai, India
Printed and bound in Great Britain

Contents

Preface

In 1990 I persuaded my colleagues in the Department of Systems Design Engineering at the University of Waterloo to approve two brave and exciting ideas. First, we'd upgrade all our boring old MSDOS computers to NeXT machines. NeXT machines were black, beautiful, and soon-to-be discontinued. We all enjoyed programming them, so the cost in terms of obsolete software when we went back to DOS/Windows computers must have been dozens of development-years. My second, rather better, idea was to put a software design course in our undergraduate programme.

Systems Design at Waterloo taught a wide-ranging curriculum in engineering, with emphasis on systems theory and design, and our students emerged as 'superb generalists'. That was our claim, and the careers of former students prove it was – and still is – true. But their exposure to software amounted to an early programming course in Pascal, then Matlab alongside Numerical Methods. In later courses they might pick up FORTRAN or C. I argued for a course distilling the insights of computer science (CS), without the depth of focused courses in a full CS curriculum, but providing key methods and tools for designing reliable, efficient, maintainable software. The new course would be broad – from data structures to software engineering – but selective, buttressed by a challenging, integrative software project.

I wanted to teach software design because I love to program, I've done a lot of it on medium-sized industry projects, and I had something to say about methodology. As an engineer I like systematic methodology, but as a scientist I've been taught to look for evidence, and there seemed to be surprisingly little evidence that the design processes and techniques in software engineering are optimal. To convey a suitable scepticism as well as give worthwhile advice and promote good practice was a challenge I wanted to tackle.

The department approved a new core course in software design and I taught it for the following five years. The framework of my lectures and notes was: introduce with deduction, support with induction and promote with abduction. First, the class would look at software from a particular perspective, highlight the features that are prominent in that way of seeing things, and deduce some rules for design (deduction). Next we'd review the experimental evidence testing those rules, and in most cases supporting them, and demonstrate their application in practical exercises (induction). Finally, I'd turn up the polemic, using anecdote and the voice of experience to promote good practice, and tales of doom of what happens when you neglect the rules. (Abduction means reasoning by analogy from similar cases. It also might mean kidnapping a person for a particular cause – in this case, good software design.) This framework was, I thought, a refreshing

change from design approaches that merely tell you what to do and don't say why (maybe because they don't know). But engineers can only take so much scientific scepticism and balance. They want to get on and make things, so the more searching inductive parts of the course gradually gave way to added advice based on the collective experience of programmers.

This book grew out of that course, and still reflects it. Chapter 2, the book's core, deduces principles and rules from nine perspectives on what software is. *Software Design for Engineers and Scientists* is unusual among books on software and programming in laying this foundation. The chapters on applied software design include reflection on the empirical (inductive) evidence for how good methods are. But the majority of the text is advice, supported inductively and abductively by numerous examples. Much of this is straightforward information – on writing C++ or on the details of algorithms, for example. But some advice is open to question, so I hope the book also conveys how to challenge and test received ideas about software design.

After I left Waterloo, I continued to develop the material that's now this book. It has been tried, tested, augmented and enhanced in courses at Memorial University of Newfoundland, Canada, and the University of York, UK. Numerous graduate students have learned software design through it, and provided candid, valuable feedback. But the world of software design has changed significantly since the early drafts. Three important things have happened that affected the shape of the book:

Important thing number 1: The tension between programming as craft and traditional software engineering has been sharpened by the introduction and advocacy of development paradigms like Extreme Programming and Agile Programming. Extreme Programming doesn't just challenge traditional assumptions, it overturns them. As a revolutionary approach, it is promoted with confident, sometimes bellicose rhetoric. But some of the big ideas of Extreme Programming are as lacking in empirical support as those of traditional software engineering. There's no shortage of anecdotes about cutting development time and happy programmers, but controlled experiments on design methodology are so difficult to do that we really don't know the true benefits and costs yet. What's needed is to identify, or at least suggest, the best elements of the competing paradigms. This I try to do here; admittedly not through controlled experiments, but as a practising programmer, I take all the advice I can get and test it on real projects. This book shares what I've learned.

An aside: because it contains so many code examples in the context of the whole software design process, this book could claim to be the first Extreme Programming text that really gives code the status it should have – at the centre. Unfortunately it can't claim that, because it doesn't buy into the whole Extreme Programming package.

See page 2 for more on this controversy.

Important thing number 2: The rise of the web has meant that students are increasingly familiar with getting computers to do what they want via an artificial language. Although their practical exposure

may amount only to HTML tweaking, the upshot is they are able to launch into learning a 'difficult' programming language like C++ without a prior, gentle, introduction to programming and computers in general. Consequently *any* science or engineering student ought to be able to use this book, even if they have not had a first programming course. To ensure this is possible, the tutorial introduction to C++ in Chapters 4 and 5 is selective, but at first gentle.

Important thing number 3: The programming language landscape has changed. Java emerged in the mid-1990s and is dominant in many fields. C# could be the language of the future, at least for Windows-based applications. The language used for this book, C++, has been standardized. This environmental change has affected both the text and the program examples, and it has meant that I will be providing code in other languages on the companion website. Chapter 4 includes some comments about the trade-offs between languages and idioms (page 51).

The testimonials I've had from Waterloo students who still use their course notes suggest that the original idea of software design for engineers (and scientists) was a good one – perhaps even good enough to compensate for the NeXT machine debacle. I think it still is.

Acknowledgements

Software Design for Engineers and Scientists explains how many different perspectives there are on what software is, and how we can learn from them all. But there's no doubt that the greatest benefit to a designer comes from working with people who are better designers. I've been lucky to work alongside some great programmers and learn from them. Chronologically, rather than in order of importance (which I couldn't even estimate), these are the people who have left a permanent stamp on my thinking and ability as a software designer: Jürgen Foldenauer, Tim Dennis, Chris Toulson, Guy Vonderweidt, Charles Nahas, Lawrence Croft, Michael Chambers, Jason Fischl, Steffen Lindner, Mehran Farimani, Li-Te Cheng. Thank you all – wherever you are.

My Waterloo students had to provide feedback on the notes that became this book. Among those writing helpful critiques, the most important was Todd Veldhuizen, and I also learned a lot from Don Bowman. Many graduate students read later drafts and provided their corrections and comments. Thanks to my colleagues at Waterloo for approving the software design course, particularly Ed Jernigan, Shekar, and the TAs who were so encouraging about the course when the project was driving the students (and us) mad.

Thanks to my colleagues in the Electrical and Computer Engineering discipline at Memorial for providing the most supportive working environment I've experienced: particularly (in the context of this book) to Theo Norvell for letting me pontificate on software design during his courses, and to John Quaicoe for wise management and inspiring teaching quality.

Thanks to the staff and students of the Department of Electronics at York, especially Stuart Porter and the students in Data Structures and Algorithms who have experienced some of this book as it went through yet more iterations.

Finally, my thanks to Gill, Luke and Stephen for everything else.

Errors

Errors are my responsibility. If you find them I'd like to know, and I'll ensure that you're acknowledged in any future edition. You can contact me by writing to the publisher.

I'm happy to hear about any kind of error, from missing evidence about the merits of a particular method, to program bugs. However, there is one kind of code error that you don't need to tell me about.

```
char buf[80];
cin >> buf;
```

probably is a bug, whereas

```
string buf;
cin >> buf;
```

probably isn't. By the end of Chapter 6, all readers will know that. But I've left things like the first example in Chapters 4 and 5 (suitably flagged as bug-spotting exercises), because they are classic examples of a programming error you *must* grapple with to be forearmed against (see page 174).

1 Introduction

1.1 Theme

This book is about:

- how to design good software
- how to program in C++
- data structures and algorithms
- scientific and engineering programming.

The book is modular but integrated. On its four themes it says both less and more than specialized texts.

- Software design is not only a big subject, it's also fast-changing and controversial. Current debates include: traditional software engineering versus extreme programming, UML versus ad hoc notation, C++ versus Java versus C#. This book outlines the issues, summarizes advice from both sides, then plumps for a particular approach to show practical real examples. A lot gets left out. But by starting with *why* we design software the way we do, the book doesn't just prescribe, it explains too.
- C++ is a big language now. Some people say the way to learn is by full immersion in the standard library. We take a much more modest and traditional approach. The book could, in principle, have featured Java, C#, or some other language (but see page 51). C++ is here, with a tutorial, because the essence of software is the program code, so a book like this has to give a central place to real programs in a real programming language.
- Data structures and algorithms are a fundamental and relatively unchanging part of software design, so we need to talk about them, giving lots of standard examples. But full coverage would demand a full book and there are plenty of good ones already. We focus on how the software designer uses data structures and algorithms to solve practical problems.
- This book includes a few recipes for scientific programming: case studies in Chapters 15 to 17, and examples of programs elsewhere that might be useful for practical science and engineering applications. But I hope you'll leave the book as a designer, not just with more knowledge about particular applications. Indeed, one message of the book is to treat recipes with care. Although Chapter 2 includes the rule 'Exploit existing solutions', it also emphasizes that 'Every program has surprises'. Uncovering and understanding any surprises in existing solutions, including other people's software recipes, is a vital skill.

The book's overriding purpose is to help you to think and act as a good software designer.

1.2 Audience

Computer programs manage society, enable every kind of telecommunication, mediate almost all impersonal business tasks. In engineering and science, they conduct experiments, analyse data, simulate complex phenomena and control complex systems. Every engineer and scientist can understand the way these programs are designed. *Software Design for Engineers and Scientists* is written to explain how. It brings together important ideas about what software is and how it is made, building from fundamentals to principles and practices. It overviews all the important aspects of software technology and the key steps in design. It shows how to program well, design substantial pieces of software correctly and efficiently, and apply the tools of computer science in an effective way. It provides copious examples and exercises to help you learn by doing. And if, having read it, you want to pursue further directed learning in computer science, the book provides a broad and reliable foundation for advanced software studies.

Software design also provides opportunities for creative problem solving, elegant crafting, and the conversion of imaginative ideas into real systems – it is stimulating and enjoyable, as well as important. As author, I have to come clean and admit that I'm excited by the subject: I enjoy writing programs and I want to make them as good as possible. I'm not in the business of trying to convert you, but I hope you'll capture at least some of the delight of software design through this book!

1.3 Three definitions and a controversy

In this book three terms appear many times: *software design*, *software engineering* and *programming.*

Software design means everything to do with creating a computer program for a particular purpose. It includes any needs analysis, specification, high-level and low-level design, modularization, coding (i.e. writing a program in a programming language), integration, debugging, testing, verification and validation, maintenance. It isn't necessary to worry about all those steps right now, just to know that *software design* encompasses them. Now comes a terminological difficulty: the term *software engineering* also includes all the steps, and, depending on who you talk to, *programming* does too! There are differences of meaning, but they aren't always shared by different people. So here is an explanation of how the three terms are used in this book.

Software design itself is a generic term with no implication about scale or standards. It applies to the whole process of creating the smallest 'Hello, world' program and to the whole process of creating a 20 million line operating system.

Software engineering means using systematic methodologies in the design process. It emphasizes that the non-coding steps of design become more important as the project gets larger and it draws on other engineering disciplines (and management) for planning techniques, processes, and standards that organize and control those steps. It also applies to coding techniques that enforce good practice and protection against errors. *Software engineering* takes the big picture and looks inwards, with coding as just one step in the process.

Programming looks outwards from coding and sees planning, specification, testing, etc. all through the lens of the program text. This

is not to say that programming is *just* coding, but that is its core, and many perspectives on programming use the coding process as the organizing mechanism for all other parts of the design.

If this all seems a little abstract, let's see how *software engineering* and *programming* contrast. A 'software engineer' might criticize 'programming' by saying that code-centric design does not put enough emphasis on being defensive, allowing safety factors in performance estimation, and considering the implications of specification changes. For example, many books have been written on the analysis and design of data structures and algorithms. But some forget to tell the reader the most certain fact about a specification: it will change. (Chapter 13 of this book demonstrates how completely proper reasoning from a specification to a data structure and algorithm implementation can be blown apart by a minor change to the specification.) To a 'software engineer', this sort of oversight typifies the danger of focusing on the code.

The Association for Computing Machinery (the professional body that represents programmers in the USA and develops curriculum guidelines for computer science programs) recommends that software engineering concepts be taught even in the earliest programming courses. One valuable consequence is an emphasis on good testing methods (which everyone agrees are important). One questionable consequence is the early introduction of loop invariants, which then tend to be forgotten (see page 14).

On the other hand, a 'programmer' might protest that some approaches to software engineering neglect coding all together. At best, they emphasize what a small part coding has in the total software development task. At worst, they give the impression that the important jobs in software system design can be done without knowing anything of the code-writing dirty work. And yet the code is the final essential product – the place where the difference between good and bad design really counts.

Since the mid-1990s, *programming* has been challenging the traditional *software engineering* view by providing, through methodologies like extreme programming, a framework for the design of big systems in which the code is central. (This challenge could be seen as redefining software engineering to encompass code-centric views, but I am going to stick with using *software engineering* in the traditional way.) Software engineering promotes systematic organization at every level of a project, but now programming (bearing uncompromising tags like *extreme*) is blatantly advocating design in ways that seem to traditionalists chaotic and unmanageable. This book flags the major tensions, and tries to find insights from both sides wherever possible. Software engineering is made essential to programming by treating modularization, data structure design and program style from a project-wide problem-solving viewpoint. Programming is made essential to software engineering by a high view of the code, summed up in the 'textual principle' that the essence of the software product is in the source code text. Particular practices are compared, selected or rejected on their own merits.

The extreme programming FAQ at www.jera.com/techinfo/xpfaq.html gives a summary of the extreme philosophy and methods.

1.4 Essential software design

Software Design for Engineers and Scientists is pragmatic when it comes to deciding between traditional software engineering and extreme programming. But it does have an underlying view on the fundamentals. The book is unusual in the central role it gives to a way of looking at software pioneered by noted computer scientist, Fred Brooks. In the April 1987 issue of *IEEE Computer* magazine, Brooks wrote an article which has become a classic of software design: 'No silver bullet: essence and accidents of software engineering'. Brooks said that to find good strategies for designing software, we must first understand

Objects, components, or what?

One of Brookes' arguments was that, for all the hype, object-oriented programming (OOP) is not a silver bullet – it doesn't change the essential problems of software design. Many people disagree. OOP does two very important things: it provides a strong model for modularity, and it allows old code to call new code. These two mean that OO code is reusable more effectively than other kinds of code. But Brookes is probably right: even together these don't make a silver bullet to slay the software werewolf.

what software really is. His own understanding of software's essential nature, and the implications for design, are summarized in the organic software section of Chapter 2 (page 18). But the most important insight of Brooks' paper is that 'How should we design software?' only makes sense if we first ask 'What *is* software?' Brooks inspired the underlying philosophy of this book: principles, rules, practices, methods, application should all be built on an understanding of software's essential properties. This paradigm is called *essential software design* – design that engages with the essence of software.

It turns out that there are many perspectives on what software is. We will examine nine in detail. For each, we begin with an appropriate definition for 'program', review the development of insights from that perspective, and derive principles about the nature of software and rules for its design. Although this approach does not identify a single essence for software, it demarcates the subject. Everything that comes after is built on this integrated understanding of the nature of software. And where there are controversies about methods and design choices, we refer back to the principles and rules to inform our decision.

1.5 Outline of the book

The structure of the book is illustrated in Figure 1.1. The earlier chapters are lower in the diagram because they form the foundation for what comes after.

Foundations

Chapter 2 is the book's foundation, where principles about the nature of software and appropriate design rules are derived from nine different perspectives. Some of the rules are about accelerating your development as a software designer, and these are discussed in Chapter 3. The rest of the book is about the construction of programs based on the two legs of technology and application. Methods and practices are developed based on the core set of principles and rules, which themselves rest on the definitions or understandings of what software is.

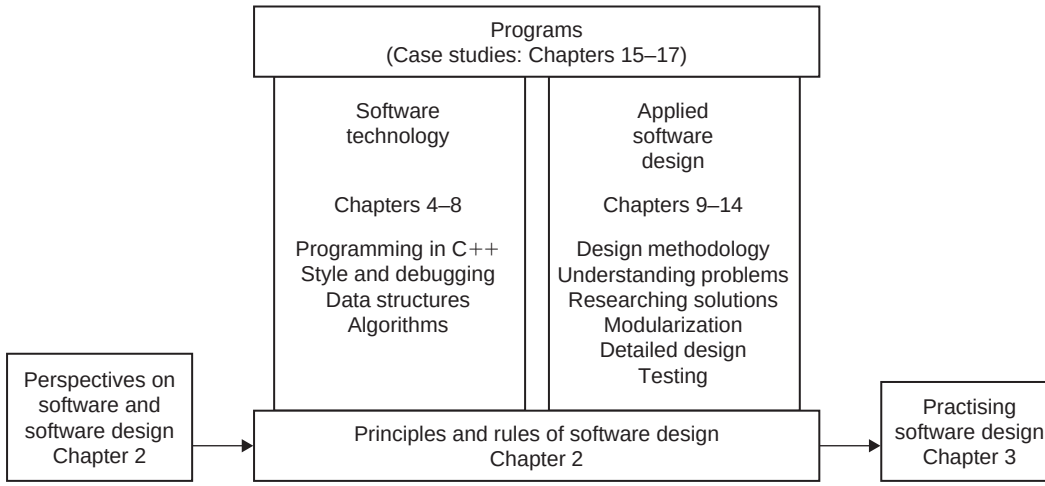

Figure 1.1 *Structure of the book*

Software technology

Accident is used here in the technical sense of those attributes that are not fundamental to a thing's existence. Some people have felt that software is just one big accident. That is a different meaning of accident … and probably not true.

One of the principles developed in Chapter 2 guides the book's approach to program examples. The choice of programming language is an *accident* rather than an *essential* of the design. But 'The textual principle' says that the program code (in whatever language it happens to be written) is the essence of software. It would be contradictory therefore not to have profuse code examples in a book on software design. And why not give real programs in a real language? C++ is therefore introduced in Chapters 4 and 5 and used heavily in examples thereafter. C++ was chosen because of its wide availability and importance. Indeed, it is arguable that another principle, 'Embrace standards', is served by using C++.

Many of C++'s foremost practitioners advocate learning it in a way that emphasizes the full range of its standard capabilities. The kind of C++ taught here is much closer to the machine than that taught, for example by Stroustrup and Deitel. The reasons for this are discussed in Chapter 4.

Chapters 4 and 5 provide a tutorial introduction to C++. They can be used in front of a computer with a C++ compiler running, and the compiler documentation at hand. They do not document all the details of C++, and they are not big enough to be a complete survey, but they will equip you to program intelligently in the language. Chapter 6 discusses program structure, style and debugging. In particular it surveys the most common programming errors, and gives advice about program clarity and how to split a program into files.

Chapters 7 and 8 are about low-level program design. Chapter 7 introduces elementary data structures, and by looking at Abstract Data Type (ADT) implementation, raises the issues in choosing data structures and algorithms. To provide tools for analysing implementation alternatives, Chapter 8 discusses algorithm analysis. On the way it gives C++ implementations for popular searching and sorting algorithms.

Applied software design

Chapters 9 to 14 cover the steps in software problem solving. Chapter 9 introduces design methodology with generic design, and develops a minimalist methodology for software. It includes overviews of design team organization and documentation for medium-sized projects. It also includes a sceptical critique of some solution generation methods, to alert you to open questions in problem solving.

Chapter 10 is about defining and understanding problems. It gives a simple recipe for writing a problem statement. The problem domain is understood by background research. This is task specific but there are some generally useful procedures that help in learning as much as possible about the problem domain. The chapter includes material on understanding human users, since they are the common element of most major software systems.

Chapter 11 deals with researching possible solutions to problems. It provides advice on quantitative analysis, experiment and simulation, then introduces diagrammatic notation techniques. Concentrating on state diagrams and statecharts, it shows how these characterize event-driven systems economically, and lead to good event-driven code. It also explains how state sketching can be useful in notating and understanding ordinary algorithms and procedures.

Modularization is increasingly important as program size grows. The treatment in Chapter 12 is founded on the principle of information hiding, and applied to both medium-sized and small programs. It includes discussion of object-oriented design and the use of state diagrams.

Chapter 13 on detailed design uses the insights developed in the Technology Chapters (4–8) to show how higher level representations (problem statement, specification, statechart, modularization description) can be turned into code. It includes discussion on ADT implementation and an example design illustrating how a good specification can lend its structure directly to a program.

Chapter 14 discusses testing, both static and dynamic. Code inspection, white box and black box testing are included. The final section of this chapter deals with the final section of the design methodology – release of the product.

Case studies

Three case studies are included at the end of the book.

Chapter 15 considers a signal-processing application – median filtering. Signal processing crops up in many data analysis applications, and median filters are handy for signal conditioning and enhancement. This is, however, a case study, not a recipe, and its purpose is to show how the steps of design fit together in a practical example that is typical of research computing.

The second case study in Chapter 16 tackles another scientific application: multidimensional optimization. Again, we'll see how the program develops as the problem and the method are better understood.

The final case study (Chapter 17) illustrates the development of a text table class that is useful for analysing the data output from experiments or simulations but can also be used more generally for manipulating textual databases. In this example, the growth of the program at many stages is illustrated, showing the interplay of specification, coding and testing. Some of the intermediate code presented contains bugs. But the finished version is heavily tested and is believed to be bug free.

1.6 Presentation conventions

There are five different styles of presentation for textual material in this book.

The running text in Times Roman font is tutorial. It is the backbone that carries the other types of material.

Program examples are in Rockwell font like this and illustrate the concepts being developed in the running text.

Some programs include line numbers like this:

```
1:  // Null program to illustrate line numbering
2:  int main( ) { return 0; }
```

The line numbers are not part of the text of the program.

Table 1.1 *An example table*

	Presentation type	**Purpose**
	Table	To supplement the running text
⇒		An arrow symbol, located in the left-hand column of a table shows the most used or most important cases. Not all tables have the arrow symbol column.

An explanatory principle or a design rule

A paragraph in Helvetica like this is a statement of a Principle or a Rule. There are 13 principles and 21 rules, all developed in Chapter 2. Later in the book, they are placed as flags to show which principle is being developed or which rule applies.

Exercises are shown in boxes like this one. They are placed with the material they exercise. You are encouraged to do the exercises as you go along. Outline solutions are sometimes given at the ends of chapters.

1.7 Chapter end material

Bibliography

All sources alluded to in this chapter, including extreme programming texts and Brooks' *Silver bullet* paper are included in the references list for Chapter 2.

A disaster

With the book deadline only weeks away, a research idea diverted me from its demanding, accusing, unfinished chapters. Immediately I wanted to test a new extension of work I'd done just six months ago. So I pulled up the programs to work out where to add new code. I scanned files, functions, class definitions and comments, written by me, for work that I know intimately. After searching analysis, I expressed my understanding of the old code in the following comprehensive thought: 'What?'

Six months ago, just like today, I had a great idea and the important thing was implementing it and testing it. In a flurry of creative activity, I did good science with a program that grew big, ugly and poorly documented. It was fully comprehensible to me then, because it was growing under my hands, urgent and vital. Trouble is, six months on, it's unreadable.

I had no excuse. First, I've been writing programs for a long time, and some of my best are widely used, read and even (among people who like that sort of thing) admired. But second, and more embarrassingly, I'd already written (and taught) drafts of this book where I'd given lots of advice about designing software. If I'd heeded it, my research code would have been much easier to read and modify.

How could I have avoided going wrong? This book will tell you. But with that kind of record, am I the right person to offer advice? Strangely enough, yes. For although I'm responsible for the software disaster I've just confessed, for ignoring my own best advice, I *had* followed other advice that allowed me to get my old code into comprehensible, usable, shape in ninety minutes. Nothing to be proud of, but not that much of a disaster after all.

The advice I hadn't followed was to do with finishing and documentation. But the advice I *had* followed was systematic construction of test cases and logbook writing. I was able to decipher my old code by going through the tests I developed and reading the contents of test logs against their timestamps. This allowed me to trace the evolution of features and recall my thought processes.

Reasoning backwards from the results of testing is not the recommended way to interpret an old program. But the moral is that even if we're not the great programmers we could be all the time, good habits give us safety nets. This book will not make you a perfect programmer, but will guide you towards habits and practices that, even when you're imperfect, like me, help you to be better.