Preface

Version 25/04/00 10:10 - 2

Once upon a time computer memory was one of the most expensive commodities on earth, and large amounts of human ingenuity were spent trying to simulate supernova explosions with nothing more than a future Nobel prize winner and a vast array of valves. Nowadays many people have enough computer memory to support simulating the destruction of most of the galaxy in any one of their hand-held phones, digital diaries, or microwave ovens.

But at least two things have remained constant throughout the history of computing. Software design remains hard [Gamma et al 1995], and its functionality still expands to fill the memory available [Potter 1948]. This book addresses both these issues. Patterns have proved a successful format to capture knowledge about software design; these patterns in particular tackle memory requirements.

As authors we had other several additional aims in writing this book. As patterns researchers and writers we wanted to learn more about patterns and pattern writing, and as software designers and architects we wanted to study existing systems to learn from them. In particular:

- We wanted to gain and share an in-depth knowledge of portable small memory techniques; techniques that work in many different environments.
- We wanted to write a complete set of patterns dealing with one single force — in this case, memory requirements.
- We wanted to study the relationships between patterns, and to group and order the patterns based on these mutual relationships, and lastly:
- We wanted an approachable book, one to skim for fun rather than to suffer as a penance.

This book is the result. It's written for software developers and architects, like ourselves, whether or not you may happen to be facing memory constraints in your immediate work.

To make the book more approachable (and more fun to write) we've taken a light-hearted slant in most of our examples for the patterns, and Duane Bibby's cartoons are delightfully frivolous. If frivolity doesn't appeal to you, please ignore the cartoons and the paragraphs describing the examples: the remaining text is as rigorous as we can make it.

This book is still a work in progress. We have incorporated the comments of many people, and we welcome more. You can contact us at our web site, `http://www.smallmemory.com/`

Dedication

To Julia and to Katherine.

Who have suffered long and are kind.

Acknowledgements

No book can be the work of just its authors. First, we need to thank John Vlissides, our indefatigable series editor: we still have copies of the email where he suggested this mad endeavour, and we're grateful for his many comments on our patterns, from the original EuroPLoP paper to the final drafts. Second, this book would not be the same without Duane Bibby's illustrations, and we hope you like them as much as we do.

We take the blame for this book's many weaknesses, but credit for most of its strengths in this book goes to the those members of the Memory Preservation Society (and fellow travellers) who took the time to read and comment on drafts of the patterns and the manuscript. These people include but are not limited to John Vlissides (again), Paul Dyson, Linda Rising, Klaus Marquardt, and Liping Zhao (EuroPLoP and KoalaPLoP shepherds for these patterns), Tim Bell, Jim Coplien, Frank Buschmann, Alan Dearle, Martine Devos, Martin Fowler, Nick Grattan, Neil Harrison, Benedict Heal, David Holmes, Ian Horrocks, Nick Healy, Dave Mery, Matt Millar, Alistair Moffat, Eliot Moss, Alan O'Callaghan, Will Ramsey, Michael Richmond, Hans Rohnert, Andreas Rüping, Peter Sommerlad, Laurence Vanhelsuwe, Malcolm Weir, and the Software Architecture Group at the University of Illinois and Urbana-Champagne, including: Federico Balaguer, John Brant, Alan Carrol, Ian Chai, Diego Fernandez, Brian Foote, Alejandra Garrido, John Han, Peter Hatch, Ralph Johnson, Apu Kapadia, Aaron Klish, An Le, Dragos-Anton Manolescu, Brian Marick, Reza Razavi, Don Roberts, Paul Rubel, Les Tyrrell, Roger Whitney, Weerasak Witthawaskul, Joseph W. Yoder, and Bosko Zivaljevic.

The team at Addison-Wesley UK (or Pearson Education, we forget which) have been great in dealing with two authors on opposite sides of the globe. We'd like to thank Sally Mortimore (for starting it all off), Allison Birtwell (for finishing it all up), and Katherin Elkstrom (for staying the distance). Credit goes also to two artists, George Platts for suggesting illustrations and Trevor Coard for creating many of them.

Finally, we must thank all the members of the patterns community, especially those who have attended the EuroPLOP conferences in Kloster Irsee. We are both admirers of the patterns 'literature' (in the same way one might be might be a fan of science fiction literature) and hope this collection of patterns will be a worthy contribution to the cannon.

Contents

Introduction

"Small is Beautiful"
E. F. Schumacher

"You can never be too rich or too thin"
Barbara Hutton

Designing small software that can run efficiently in a limited memory space was, until recently a dying art. PCs, workstations and mainframes appeared to have exponentially increasing amounts of memory and processor speed, and it was becoming rare for programmers even to need to think about memory constraints.

At the turn of a new century, we're discovering an imminent market of hundreds of millions of mobile devices, demanding enormous amounts of high-specification software; physical size and power limitations means these devices will have relatively limited memory. At the same time, the programmers of Web and database servers are finding that their applications must be memory-efficient to support the hundreds of thousands of simultaneous users they need for profitability. Even PC and workstation programmers are finding that the demands of video and multi-media can challenge their system's memory capacities beyond reasonable limits. Small memory software is back!

But what is small memory software? Memory size, like riches or beauty, is always relative. Whether a particular amount of memory is small or large depends on the requirements the software should meet, on the underlying software and hardware architecture, and on much else. A weather-calculation program on a vast computer may be just as constrained by memory limits as a word-processor running on a mobile phone, or an embedded application on a smart card. Therefore:

Small memory software is any software that doesn't have as much memory as you'd like!

This book is written for programmers, designers and architects of small memory software. You may be designing and implementing a new system, maintaining an existing one, or merely seeking to expand your knowledge of software design.

In this book we've described the most important programming techniques we've encountered in successful small memory systems. We've analysed the techniques as *patterns* – descriptions in a particular form of things already known to work [Alexander 1977, 1979]. Patterns are not invented, but are identified or mined from existing systems and practices. To produce the patterns in this book we've investigated the design of many successful systems that run on small machines. This book distils the essence of the techniques that seem most responsible for the systems' success.

The patterns in this book consider only limitations on memory: Random Access Memory (RAM), and to a lesser extent Read Only Memory (ROM) and Secondary Storage, such as disk or battery backed RAM. A practical system will have many other limitations; there may be constraints on graphics and output resources, network bandwidth, processing power, real-time responsiveness, to name just a few. Although we focus on memory requirements, some of these patterns may help with these other constraints; others will be less appropriate: compression, for example, may be unsuitable where there are significant constraints on processor power; paging is unhelpful for real-time performance. We've indicated in the individual patterns how they may help or hinder supporting other constraints.

The rest of this chapter introduces the patterns in more detail. The sections are as follows:

How to use this book	Suggests how you might approach this book if you don't want to read every page.
Introduction to Small Memory	Describes the problem in detail, and contrasts typical kinds of memory-constrained software.
Introduction to Patterns	Introduces patterns and explains the pattern format used in this book
The Patterns in this Book	Suggests several different ways of locating, relating and contrasting all the patterns in the book.

How to use this book

You can, of course, start reading this book at page one and continue through to the end. But many people will prefer to use this book as a combination of several things:

- A programmer's introduction to small memory software.
- A quick overview of all the techniques you might want to use.
- A reference book to consult when you have a problem you need to solve.
- An implementation guide showing the tricks and pitfalls of using common – or less common – patterns.

The following sections explain how to use this book for each of the above, and also for other more specialised purposes:

- Solving a particular strategic problem
- Academic study
- Keeping the boss happy

A Programmer's Introduction to Small Memory Software.

Perhaps you're starting as a new developer on a memory-constrained project, and haven't worked on these kinds of projects before.

If so, you'll want to read about the programming and design-level patterns that will affect your daily work. Often your major design concern will initially be class design, so start with the straightforward and fun PACKED DATA (ppp). You can then continue exploring several other SMALL DATA STRUCTURE patterns used a lot in memory-limited systems: SHARING (PPP), COPY-ON-WRITE (PPP) and EMBEDDED POINTER (PPP).

The immediate choices in coding are often how to allocate data structures, so next compare the three common forms of memory allocation: FIXED ALLOCATION (PPP), VARIABLE ALLOCATION (PPP) and MEMORY DISCARD (PPP). Equally important is how you'll have to handle running out of memory, so have a look at the important PARTIAL FAILURE (PPP) pattern, and perhaps also the simple MEMORY LIMIT (PPP) one.

Finally, most practical small memory systems will use the machine hardware in different ways to save memory. So explore the possibilities of READ-ONLY MEMORY (PPP), APPLICATION SWITCHING (ppp) and DATA FILES (PPP).

The description of each of these patterns discusses how other patterns complement them or provide alternatives, so by reading these patterns you can learn the most important techniques and get an overview of the rest of the book.

Quick Overview of all the Techniques

A crucial benefit of a collection of patterns is that it creates a shared *language* of pattern names to use when you're discussing the topic [Gamma Helm Johnson Vlissides 1995, Coplien 1996]. To learn this language, you can scan all the patterns quickly, reading the main substance but ignoring all the gritty details, code and implementation notes. We've structured this book to make this easy to do. Start at the first pattern (SMALL ARCHITECTURE, page XXX) and read through each pattern down to the first break:

❖ ❖ ❖

This first part of the pattern provides all you really need to know about it: the problem, its context, a simple example, and the solution. Skimming all the patterns in this way takes a

careful reader couple of hours, and provides an overview of all the patterns with enough detail that you can begin to remember the names and basic ideas of the patterns.

Reference for Problem Solving

Perhaps you're already working on a project, in a desperate hurry, but faced with a thorny problem with no simple answer. In this case, first consult the brief summaries of patterns in the front cover. If one or more patterns look suitable, then turn to each one and read its bullet points and 'Therefore' paragraph to see if it's really what you're after.

If that approach doesn't produce a perfect match, then use the index to look for keywords related to your problem, and again scan the patterns to see which are suitable.

If none of the patterns you've found so far are quite what you want, then have a look at the summary pattern diagram in the back cover – there may be useful patterns related to the ones you've already checked. Check the 'See Also' sections at the end of the patterns that seem most useful; perhaps one of the related patterns might address your problem.

Implementation Guide

Perhaps you've already decided that one or more of the patterns are right for you. You may have known the technique all along, although you've not thought of it as a pattern, and you've decided – or been told – to use it to implement part of your system.

In this case you can consult the full text for each specific pattern you've chosen. Find the pattern using the summary in the front cover, and turn to the Implementation section for a discussion of many of the issues you'll come across in using the pattern, and some of the techniques other implementers have successfully used to solve them. If you prefer looking at specifics like code, turn to the Example section first and then move back to the Implementation section.

You can also look at the Known Uses section – perhaps the systems will be familiar and you can find out how they've implemented the pattern, and the 'See Also' section guides you to other patterns you may find useful.

Helping Define a Project Strategy

If you're defining the overall strategy for a software project [Goldberg and Rubin 1995], you'll probably be concerned about many other issues in addition to memory restrictions. Maybe you're worried about time performance, real-time constraints, a hurried delivery schedule or a need for the system to last for several decades.

In this case turn to the discussion in appendix XXX. These concerns are called 'forces' [Alexander 1979]. Scan the chapter to identify the forces you're interested in; the sections on each force will tell you which patterns will best suit your needs.

Academic Study

Of course many people still enjoy reading books from start to finish. We have written this book so that it can also be read in this traditional, second millennium style.

Each chapter starts with the simpler patterns that are easy to understand, and progresses towards the more sophisticated patterns; the patterns that come first in a chapter lead to the patterns that follow afterwards. The chapters make a similar progression, starting with the large-scale patterns you are most likely to need early in a project (Architectural Patterns) and progressing to the most implementation-specific patterns (Memory Allocation).

Keeping the Boss Happy

Maybe you really don't care about this stuff at all, but your manager has bought this book for you and you want to retain your credibility. In this case, leave the book open face down on a radiator for three days. The book will then look as though you've read it, without any effort required on your part [Covey 1990].

Introduction to Small Memory

What makes a system small? We expect that the patterns in this book will be most useful for systems with memory capacities roughly between 50K to 10M bytes total, although many of the patterns are frequently used with much smaller and even much larger systems.

Here are four different kinds of projects that can have difficult meeting their memory requirements, and consequently can benefit from the patterns in this book:

1. Mobile Computing

Palmtops, pagers, mobile phones, and similar devices are becoming increasingly important. Users of these mobile machines are demanding more complex and feature-ridden software, ultimately comparable to that on their desktops. But, compared to desktop systems, a portable device's hardware resources, particularly memory, are quite limited. Because of their ubiquitous nature [Norman 1998] these machines also need to be more robust than desktop machines — a digital diary with no hard disk cannot be restarted without loosing its data.

Developments for such machines must take far more care with memory constraints than in similar applications for PCs and Workstations. Virtually all the patterns in this book may be relevant to any given project.

2. Embedded Systems

A second category of physically small device is embedded systems such as process control systems, medical systems and smart-cards. When a posh new car can have more than a hundred microprocessors in it, and with predictions that we'll all have several embedded microprocessors in our bodies within the next decade, this is a very important area.

Embedded devices are limited by their memory, and have to be robust, but in addition they often have to meet hard real-time processing deadlines. If they are life critical or mission critical they must meet stringent quality control and auditing requirements too.

In systems with memory capacity is much below about 50 Kbytes, the software must be tightly optimised for memory, and typically must make drastic tradeoffs of functionality to fit into the available memory. In particular, the entire object-orientated paradigm, though possible, becomes less helpful as heap allocation becomes inappropriate. When implementing systems below 50K, the Allocation and Data Structure patterns are probably the most important.

3. Small Slice of a Big Pie

Many ostensibly huge machines — mainframes, minicomputers or PC servers— can also face problems with memory capacity. These very large machines are most cost-effective when supporting hundreds, thousands, or even hundreds of thousands of simultaneous sessions. Even though they are physically very large, with huge physical memory capacities and communication bandwidths, their large workloads often leave relatively modest amounts of memory for each individual session.

For example, most Java virtual machines in 2000 require at least 10Mb of memory to run. Yet a Java-based web server may need to support ten thousand simultaneous sessions. Naively replicating a single user virtual machine for each session would require a real hardware server with 100 Gigabytes of main memory, not counting the memory required for the application on each virtual machine. The patterns in this book, particularly the Data Structure patterns, can increase the capacity of such servers to support large numbers of users.

4. Big Problems on Big Machines

In single-user systems with a memory capacity greater than 10 Mbytes, memory is rarely the major concern for most applications, but general-purpose computers can still suffer from limited memory capacity.

For example, organisations may have a large investment in particular hardware with a set memory capacity that it is not feasible to increase. What happens if you're a bank with twenty thousand three-year old PCs sitting on your tellers' desks and would like to upgrade the software? What happens if you bought a new 1 Gigabyte server last year, can't afford this year's model, but need to process 2 Gigabytes this year? Even if you could afford to upgrade the machines, other demands (such as your staff bonus) may have higher priority.

Alternatively, you may be working on an application that must handle very large amounts of data, such as multi-media editing, video processing, pattern recognition, weather prediction, or maintaining a collection of detailed bitmap images of the entire world. Any such application could easily exhaust the RAM in even a large system, so you'll need careful design to limit its memory use. For such applications, the Secondary Storage and Compression patterns are particularly important.

Ultimately, no computer can ever have enough memory. Users can always run more simultaneous tasks, process larger data sets, or simply choose a less expensive machine with a lower physical memory capacity. In a small way, every machine has small memory.

Types of Memory Constraint

Imagine you're just starting a new project in a new environment. How can you determine which memory constraints are likely to be a problem, and what types of constraint will give the most trouble?

Hardware Constraints.

Depending on your system, you may have constraints on one or more of the following types of memory:

RAM Memory	Used for executing code, execution stacks, transient data and persistent data.
ROM Memory	Used for executing code and read-only data.
Secondary Storage	Used for code storage, read-only data and persistent data.

You may also have more specific constraints: for example stack size may be limited, or you may have both dynamic RAM, which is fast but requires power to keep its data, and static RAM, which is slower but will keep data with very little power.

RAM is usually the most expensive form of memory, so many types of system keep code on secondary storage and load it into RAM memory only when it's needed; they may also share the loaded code between different users or applications.

Software Constraints

Most software environments don't represent their memory use in terms of main memory, ROM and secondary storage. Instead, as a designer, you'll usually find yourself dealing with heap, stack and file sizes. Table XXX below shows typical attributes of software and how each maps to the types of physical memory discussed above.

Attribute	Where it Lives
Persistent data	Secondary storage or RAM.
Heap and static data	RAM
Code Storage	Secondary storage or ROM
Executing Code	RAM or ROM
Stack	RAM

Different Types of Memory-Constrained System

Different kinds of systems have different resources and different constraints. Table 3 describes four typical kinds of system: embedded systems, mobile phones or digital assistants, PCs or workstations, and large mainframe servers. This table is intended to be a general guide, and few practical systems will match it exactly. An embedded system for a network card will most certainly have network support, for example; many mainframes may provide GUI terminals; a games console might lie somewhere between an embedded system and a PDA.

	Embedded System	Mobile Phone PDA	PC, Workstation	Mainframe or Server Farm
Typical Applicat-ions	Device control, protocol conversion, etc.	Diary, Address book, Phone, Email	Word processing, spreadsheet, small database, accounting.	E-commerce, large database applications, accounting, stock control.
UI	None.	GUI; libraries in ROM	GUI, with several possible libraries as DLLs on disk	Implemented by clients, browsers or terminals
Network	None, Serial Connection, or Industrial LAN	TCP/IP over a wireless connection	10MBps LAN	100 MBps LAN
Other IO	As needed – often the main purpose of device	Serial connections	Serial and parallel ports, modem, etc.	Any, accessed via LAN

Table 3: Comparison of different kinds of system

All these environments will normally keep transient program and stack data in RAM, but differ considerably in their other memory use. Table XXX below shows how each of these kinds of system typically implements each kind of software memory.

	Embedded System	Mobile Phone, PDA	PC, Workstation	Mainframe or Server Farm
Vendor-supplied Code	ROM	ROM	On disk, loaded to RAM	Disk, loaded to RAM

3rd Party Code	None	Loaded to RAM from flash memory	As vendor-supplied code	As vendor-supplied code
Shared Code	None	DLLs shared between multiple applications	DLLs shared between multiple applications	DLL and application code shared between multiple users
Persistent Data	None, or RAM	RAM or flash memory.	Local hard disk or network server.	Secondary disk devices.

Table 1: Memory Use on each Type of System

Note how mobile phones and PDAs treat third-party code differently from vendor-supplied code, since the former cannot live in ROM.

Relative Importance of Memory Constraints

Table 4 shows the importance of the different constraints on memory for typical applications on each kind of system discussed above. Three stars mean the constraint is usually the chief driver for a typical project architecture; two stars mean it is an important design consideration. One star means the constraint that may need some effort from programmers but probably won't affect the architecture significantly; and no stars mean it's virtually irrelevant to development.

	Embedded System	**Wireless PDA**	**PC, Workstation**	**Mainframe or Server Farm**
Code Storage	**	**		
Code Working Set		**	*	
Heap and Stack	***	**	*	*
Persistent Data	***	*		

Table 4: Importance of Memory Constraints

In practice, every development is different; there will be some smart-card applications that can virtually ignore the restrictions on heap and stack memory, just as there will be some mainframe applications where the main constraint is on persistent storage.

Introduction to Patterns

What, then, actually *is* a pattern? The short answer is that a pattern is a "*solution to a problem in a context*" [Alexander 1977,Coplien 1996]. This focus on context is important, because with a large number of patterns it can be difficult to identify the best patterns to use. All the patterns in this book, for example, solve the same problem – too little memory – but they solve it in many different ways.

A pattern is not just a particular solution to a particular problem. One of the reasons programming is hard is that no two programming problems are exactly alike, so a technique that solves one very specific problem is not much use in general [Jackson 1995]. Instead, a pattern is a generalised description of a solution that solves a general class of problems — just as an algorithm is a generalised description of a computation, and a program is a particular implementation of an algorithm. Because patterns are general descriptions, and because they are higher-level than algorithms, you should not expect the implementation to be the same every time they are used. You can't just cut out some sample code describing a pattern in this book, paste it into your program and expect it to work. Rather, you need to understand the general idea of the pattern, and to apply that in the context of the particular problem you face.

How can you trust a pattern? For a pattern to be useful, it must be known to work. To enforce this, we've made sure each pattern follows the so-called Rule of Three: we've found at least three known uses of the solution in practical systems. The more times a pattern has been used, the better, as it then describes a better proven solution. Good patterns are not invented; rather they are identified or 'mined' from existing systems and practices. To produce the patterns in this book we've investigated the design of many successful systems that run on small machines. This book distils the essence of the techniques that seem most responsible for the systems' success.

Forces

A good pattern should be intellectually rigorous. It should present a convincing argument that that its solution actually solves its problem, by explaining the logic that leads from problem to solution. To do this, a good pattern will enumerate all the important *forces* in the context and enumerate the positive and negative consequences of the solution. A force is '*any aspect of the problem that should be considered when solving it*' [Buschmann et al 1996], such as a requirement the solution must meet, a constraint the solution must overcome, or a desirable property that the solution should have.

The most important forces the patterns in this book address are: *memory requirements*, the amount of memory a system occupies; and *memory predictability*, or whether this amount can be determined in advance. These patterns also address many other forces, however, from real-time performance to usability. A good pattern describes both its benefits (the forces it *resolves)*, and its disadvantages (the forces it *exposes)*. If you use such a pattern you may have to address the disadvantages by applying another pattern [Meszaros and Doble 1998].

The Appendix discusses the major forces addressed by the patterns in this collection, and describes the main patterns that can resolve or expose each force. There's also a summary in the table printed inside the back cover.

Collections of Patterns

Some patterns may stand alone, describing all you need to do, but many are *compound patterns* [Vlissides 1998; Riehle 1997] and present their solution partially in terms of other patterns. Applying one pattern resolves some forces completely, more forces partially, and also exposes

some forces that were not yet considered. Other, usually smaller-scale, patterns can address problems left by the first pattern, resolving forces the first pattern exposes.

Alexander organised his patterns into a *pattern language*, a sequence of patterns from the highest level to the lowest, where each pattern explicitly directed the reader to subsequent patterns. By working through the sequence of patterns in the language, an architect could produce a complete design for a whole room, building, or city [Alexander 1977].

The patterns in this book are not a pattern language in that sense, and we certainly have not set out to describe every programming technique required to build a complete system! Where practical, however, we have described how the patterns are related, and how using one pattern can lead you to consider using other patterns. The most important of these relationships are illustrated in the diagram printed inside the back cover.

A Brief History of Patterns

Patterns did not originate within programming. A Californian architect, Christopher Alexander developed the pattern form as a tool for recording knowledge of successful building practices (architectural folklore) [Alexander 1977, 1979]. Kent Beck and Ward Cunningham adapted patterns to software, writing a few patterns than were used to design user intefaces at Textronix. The 'Hillside Group' of software engineers developed techniques to improve pattern writing, leading to the PLoP series of conferences.

Gamma, Helm, Johnson and Vlissides, developed patterns for object-oriented frameworks in their 1995 book '*Design Patterns*'. Many other valuable pattern books have followed, particularly *Patterns of Software Architecture* [Buschmann 1996], *Analysis Patterns* [Fowler 1997], the Addison-Wesley *Pattern Languages of Program Design* series, and more specialist books, such as the *Smalltalk companion to Design Patterns* [Alpert, Brown and Woolf 1998], etc. You can now find patterns on virtually any aspect of software development using the *Patterns Almanac* [Rising 2000].

How We Wrote This Book

To produce this particular collection of patterns, we built up a comprehensive list of memory saving techniques that we'd seen in software, been told about, seen on the web, or just known about all along. We then pruned out any techniques that seemed related but that didn't actually save memory (such as bank switching or power management), and any techniques outside the scope of this book (UI design, project management). The result was several hundred different ideas, known uses and examples.

We then grouped them together, looking for a number of underlying themes or ideas that provided a set of underlying techniques. Each technique formed the basis for a pattern, and we wrote them up as a draft that was presented at a pattern writers workshop in 1998 [Noble and Weir 1998, Noble and Weir 2000]. As we built up a set of draft patterns, and received comments, criticism and suggestions, we 'refactored' [Fowler 1999] the patterns to give a more complete picture, expanding the scope of one pattern and reducing or changing the thrust of another. We analysed the major forces addressed by each pattern, and the relationships between the patterns, to find a good way to arrange the patterns into a coherent collection. The result is what you're reading now.

Like any story after the fact, this step-by-step approach wasn't exactly what happened in practice. The reality was much more organic and creative; it took place over several years, and most of the steps happened in parallel throughout that time: but in principle it's accurate.

Our Pattern Format

All the patterns in this book use the same format. Consider the example (overleaf/on the facing page), an abbreviated version of one of the data structure patterns. The full pattern is on page XXX.

Packed Data

Also Known As: Bit Packing

How can you reduce the memory needed to store a data structure?

- You have a data structure (a collection of objects) that has significant memory requirements.
- You need fast random access to every part of every object in the structure…

No matter what else you do in your system, sooner or later you end up having to design low-level data structures to hold the information your program needs…

For example, the Strap-It-On's Insanity-Phone application needs to store all of the names and numbers in an entire local telephone directory (200,000 personal subscribers)….

Because these objects (or data structures) are the core of your program, they need to be easily accessible as your program runs…

Therefore: *Pack data items within the structure so that they occupy the minimum space.*

There are two ways to reduce the amount of memory occupied by an object…

Consider each individual field in turn, and consider how much information that field really needs to store. Then, chose the smallest possible language-level data type that can store than information, so that the compiler (or assembler) can encode it in the minimum amount of memory space…

Considering the Insanity-Phone again, the designers realised that local phone books never cover more than 32 area codes – so each entry requires only 5 bits to store the area code…

Consequences

Each instance occupies less memory reducing the total *memory requirements* of the system, even though the same amount of data can be stored, updated, and accessed randomly…

However: The *time performance* of a system suffers, because CPUs are slower at accessing unaligned data…

❖ ❖ ❖

Figure 1: Excerpt from a Pattern

The example shows the sections of the pattern, which are as follows:

Pattern Name	Every pattern has a unique name, which should also be memorable
Cartoon	A cartoon provides an amusing visual representation of the solution.
Also Known As	Any other common names for the pattern, or for variants of the pattern.
Problem Statement	A single sentence summarises the main problem this pattern solves.
Context Summary	Bullet points summarise each main force involved in the problem, giving an at-a-glance answer to 'is this pattern suitable for my particular problem?'

Context Discussion	Longer paragraphs expand on the bullet points: when might this pattern apply; and what makes it a particularly interesting or difficult problem? This section also introduces a simple example problem – often fanciful or absurd – as an illustration.
Solution	**Therefore:** A single sentence summarises the solution.
Solution Description	Further paragraphs describe the solution in detail, sometimes providing illustrations and diagrams. This section also shows how the solution solves the example problem.
Consequences	This section identifies the typical consequences of using the pattern, both advantages and disadvantages. To distinguish the two, the positive consequences are first, and the negative ones second, partitioned by the word '**However:**'. In this discussion, the important forces are shown in *italic*; these forces are cross-referenced in the discussion in Chapter XXX.
Separator	Three '❖' symbols indicate the end of the pattern description.

Throughout the text we refer to other patterns using SMALL CAPITALS: thus 'PACKED DATA.'

This is all you need to read for a basic knowledge of each pattern. For a more detailed understanding – for example if you need to apply the pattern – every pattern also provides much more information, in the following sections:

Implementation	A collection of 'Implementation Notes' discuss the practical details of implementing the pattern in a real system. This is usually the longest section extending into several pages. The instructions in the implementation notes are not obligatory; you may find alternative, and better, ways to implement the pattern in a particular context. The notes capture valuable experience and it's worth reading them carefully before starting an implementation.
Example	This section provides code samples from a particular, usually fairly simple, implementation, together with a detailed discussion of what the code does and why. We recommend reading this section in conjunction with the Implementation section. Our web site [www.smallmemory.com] provides the full source for most of the samples.
Known Uses	This section describes several successful existing systems that use this pattern. This section validates our assertion that the pattern is useful and effective, and it also suggests places where you might look for further information.
See Also	This section points the reader to other related patterns in this book or elsewhere. This section may also refer you to books and web pages with more information on the subject or to help further with implementation.

Major Techniques and Pattern Relationships

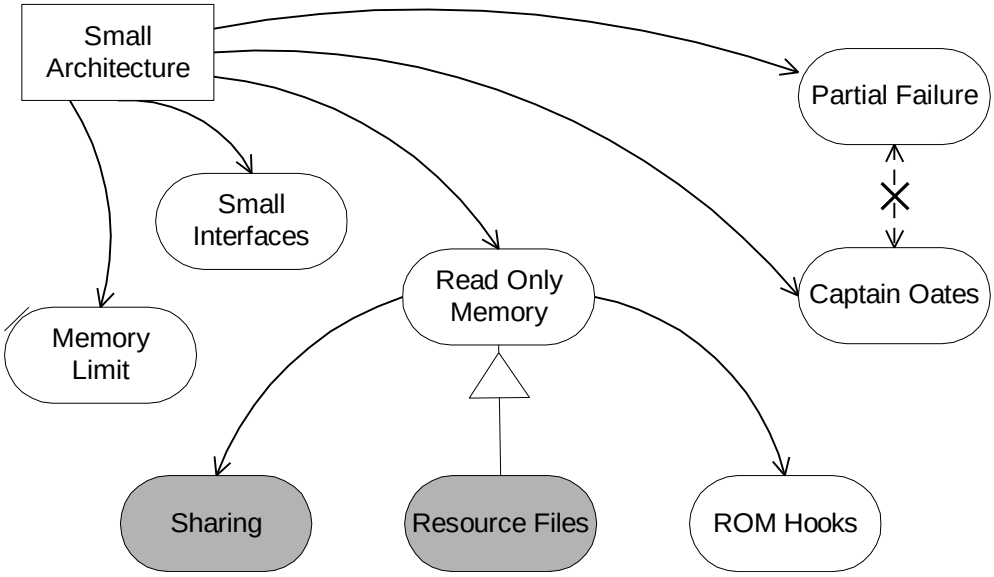
We've grouped the patterns in this book into five chapters. Each chapter presents a Major Technique for designing small memory software: Architecture, Secondary Storage, Compression, Data Structures, and Allocation.

Each Major Technique is, itself, a pattern, though more abstract than the patterns it contains. We've acknowledged that by using a variant of the same pattern format for Major Techniques as for the normal patterns.

In each Major Technique, a 'Specialised Patterns' section summarises each pattern in the chapter, replacing the 'Example' section. (see Figure XXX).

Specialised Patterns

The following sections describe six specialised patterns that describing ways architectural decisions can reduce RAM memory use. The figure below shows how they interrelate…



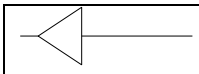
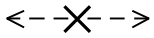
The patterns in this chapter are as follows:

SMALL INTERFACES — Design the interfaces between components to manage memory explicitly, minimising the memory required for their implementation…

Figure 2: Excerpt from a Major Technique

This diagram illustrates the relationships between the patterns in the chapter. In the diagram, the rectangle represents the Major Technique pattern; white ovals represent patterns in the chapter; and grey ovals represent patterns in other chapters. The relationships between the patterns are shown as follows [Noble 1998]:

	Uses	If you're using the left-hand pattern, you should also consider using the right-hand one. The smaller-scale pattern on the right resolves *forces* exposed by the larger pattern on the left. For example if you are using the READ-ONLY MEMORY pattern, you should consider use the HOOKS pattern.

	*Specialises*	If you are using the left hand pattern, you may want the right-hand pattern in particular situations. The right-hand pattern is a more specialised version of the left-hand one, resolving similar *forces* in more particular ways. For example, RESOURCE FILES are a special kind of READ-ONLY MEMORY.
	*Conflicts*	Both patterns provide alternative solutions to the same problem. They resolve similar *forces* in incompatible ways. For example, PARTIAL FAILURE and CAPTAIN OATES both describe how a component can deal with memory exhaustion; either by providing a reduced quality of service, or by terminating another component.

The Running Example

We've illustrated many of the patterns with examples taken from a particularly memory-challenged system, the unique Strap-It-On™ wrist-mounted PC from the well-known company StrapItOn. This product, of course, includes the famous Word-O-Matic word-processor, with its innovative Morse Code keypad and Voice User Interface (VUI).

Figure 3: The Strap-it-on

If you're foolish enough to implement any of the applications we suggest, and make money out of it, well good luck to you!

The Patterns in this Book

We've chosen one particular order for the patterns in this book (see the table in the front inside cover). This order makes the patterns easy to learn, working top-down so that the first patterns set the scenes for the patterns in later chapters.

There are many other valid ways to arrange or discuss the list patterns. This section examines from several different perspectives: the list of Major Techniques, the forces addressed by each technique, different approaches to saving memory, and via case studies. You may also like to consult the discussion '**Error! Reference source not found.**' on page XXX, which recommends patterns according to the types of small software involved.

The Major Techniques

The patterns are organised into five Major Techniques, summarised in the following table:

Small Data Structures	Defining data structures and algorithms that contrive to reduce memory use.
Memory Allocation	Mechanisms to assign a data structure from the 'primordial soup' of unstructured available memory, and to return it again when no longer required by the program.
Compression	Processing-based techniques to reduce data sizes by automatically compressing data.
Secondary Storage	Using disk, or equivalent, as an adjunct to RAM.
Small Architecture	Memory-saving techniques that require co-operation between several components in a system.

The Forces Addressed by the Patterns

Another way to look at the patterns is to compare the forces addressed by each pattern.

Chapter XXX (Forces) discusses the forces in detail, and discusses which patterns address each force. Meanwhile the following two tables provide a partial summary of that chapter. Table XXX summarises ten of the forces we consider most important, and table XXX shows how the patterns address each one.

Memory Requirements	Does the pattern reduce the absolute amount of memory required to run the system?
Memory Predictability	Does the pattern make it easier to predict the amount of memory a system will require in advance?
Real-time Response	Does the pattern decrease the latency of the program's response to events, usually by making the run-time performance of the program predictable?
Start-up Time	Does the pattern reduce the time between the system receiving a request to start the program, and the program beginning to run?
Time Performance	Does the pattern tend to improve the run-time speed of the system?
Local vs. Global	Does the pattern tend to help encapsulate different parts of the application, keeping them more independent of each other?
Secondary	Does the pattern tend to shift memory use towards cheaper secondary storage in

Storage	preference to more expensive RAM?
Maintain-ability	Does the pattern encourage better design quality? Will it be easier to make changes to the system later on?
Programmer Effort	Does the pattern reduce the total programmer effort to produce a given system?
Testing cost	Does the pattern reduce the total testing effort for the application development?

Table 2: Ten Important Forces

The following table shows how each pattern addresses these forces. If the pattern generally benefits you as far as this force is concerned ('resolves the force'), it's shown with a 'Y'. If it's generally a disadvantage ('exposes the force') that's shown with an 'N'. Appendix XXX explores these forces in much more detail.

[Typesetter to replace N in table and paragraph with sad face ☹ , and make those cells white font on black background.

Typesetter to replace Y in table and paragraph with happy face ☺, black font on white background.

Typesetter to replace O in table and paragraph with face 😐 - black font on light grey background

This table is duplicated in the inside back cover.]

		Time Performance	Real-time	Start-up time	Local vs. Global	Predictability	Quality and Maintainability	Programmer Effort	Testing cost	Usability	
Architecture					O	Y	Y	O		Y	
	Memory Limit				Y	Y			O		
	Small Interfaces	N	Y		Y	Y	Y	O	O		
	Partial Failure				N	Y	Y	N	N	Y	
	Captain Oates	N			Y	O		N	N	Y	
	Read-only Memory		Y	Y	O		N	N	Y		
	Hooks	N					Y	Y	N		
Secondary Storage			N		N			N		N	
	Application Switching	N	Y	Y	O	Y	Y	O	Y	N	
	Data Files	N		N	O	Y		N	Y	N	
	Resource Files	N		N			Y	O			
	Packages		N	Y			Y	N	N	O	
	Paging	N	N		Y		Y	Y	Y	Y	
Compression		N	O			N	N	N	N		
	Table Compression	O							N		
	Sequence Compression	O	Y			Y			N		
	Adaptive Compression	N	N					N	N		
Data Structures			O		Y	O	Y	N	O	Y	
	Packed Data	N			Y		N	N			
	Sharing	Y		Y	N		O	N	N		
	Copy-on-Write	O	N	Y		N		N	N		
	Embedded Pointer	Y	Y		N	Y	N	N			
	Multiple Representations	Y			Y	N	Y	O	N		

		Time Performance	Real-time	Start-up time	Local vs. Global	Predictability	Quality and Maintainability	Programmer Effort	Testing cost	Usability	
Allocation		O	O	O	O	Y					
	Fixed Allocation	Y	Y	N		Y		O	Y	N	
	Variable Allocation	N	N	Y	N	N	Y	Y	N		
	Memory Discard	Y	Y	Y	Y	Y		Y	N		
	Pooled Allocation	Y	Y	N		Y			N		
	Compaction	N	N					N	N		
	Reference Counting	N	Y		Y		Y				
	Garbage Collection	Y	N		Y	N	Y	Y			

Reduce Reuse Recycle

Environmentalists have identified three strategies to reduce the impact of human civilisation on the natural environment:

- Reduce consumption of manufactured products and the production of waste products.
- Reuse products for uses other than that for which they were intended.
- Recycle the raw material of products to make other products.

Of these, reduction is the most effective; if you reduce the amount of waste produced you don't have to worry about how to handle it. Recycling is the least effective; it requires a large expenditure of energy and effort to produce new finished products from old waste. The patterns we have described can be grouped in a similar way. They can:

- **Reduce** a program's memory requirements. Patterns such as PACKED DATA, SHARING, and COMPRESSION reduce the amount of absolute memory required, by reducing data sizes and removing redundancy. In addition the SECONDARY STORAGE and READ ONLY MEMORY patterns reduce RAM memory requirements by using alternative storage.
- **Reuse** memory for a different purpose. Memory used within a FIXED ALLOCATION or in POOLED ALLOCATION is generally (re)used to store a number of different objects of roughly the same type, one after another. HOOKS allow software to reuse existing read-only code rather than replacing it.
- **Recycle** memory for different uses at different types. VARIABLE ALLOCATION, REFERENCE COUNTING, COMPACTION, GARBAGE COLLECTION and CAPTAIN OATES all help a program to make vastly different uses of the same memory over time.

Case Studies

This section looks at three simple case studies, and looks at the patterns you might use to deliver a successful implementation in each case.

1. Hand-held Application

Consider working on an application for a hand-held device such as Windows CE, PalmOs or an EPOC smart-phone.

The application will have a GUI, and will need much of the basic functionality you'd expect of a full-scale PC application. Memory is, however, more limited than for a PC; acceptable maximums might be 2Mb of working set in RAM, 700Kb of (non-library) code, and a couple of Mb or so of persistent data according to the needs of each particular user.

Memory is limited for all applications on the device, including your own. The SMALL INTERFACES architectural pattern is ubiquitous and vital. Applications will use RESOURCE FILES, as dictated by the operating system style guide, and to keep code sizes and testing to a minimum you'll want to access the vendor-supplied libraries in READ ONLY MEMORY using the libraries' HOOKS. Since the environment supports many processes and yours won't be running all the time, you'll need persistent data stored in DATA FILES.

The environment may mandate other architectural patterns: PalmOs requires APPLICATION SWITCHING; CE expects CAPTAIN OATES; and EPOC expects PARTIAL FAILURE. If you're not working for the system vendor then yours will be a 'third party' application loaded from secondary storage, so you may use PACKAGES to reduce the code working set.

Most of your application's objects will use VARIABLE ALLOCATION or MEMORY DISCARD. Components where real-time performance is important –a communications driver, for example – will use FIXED ALLOCATIONand EMBEDDED POINTERS.

Classes that have many instances may use PACKED DATA, MULTIPLE REPRESENTATIONS, SHARING or COPY-ON-WRITE to reduce their total memory footprint. Objects shared by several components may use REFERENCE COUNTING.

2. Smart-card Project

Alternatively, consider working on a project to produce the software for a smart-card – say a PCMCIA card modem to fit in a PC. Code will live in ROM (actually flash RAM used as ROM), and there's about 2Mb of ROM in total, however there's only some 500K of RAM. The only user interface is the Hayes 'AT' command set available via the serial link to the PC; the modem is also connected to a phone cable.

The system code will be stored in the READ-ONLY MEMORY, along with static tables required by the modem protocols. You'll only need a single thread of control and a single, say 50K, stack.

The real-time performance of a modem is paramount, so most long lived objects will use FIXED ALLOCATION. Transient data wil use MEMORY DISCARD, being stored on the stack. The system will need lots of buffers for input and output, and these can use POOLED ALLOCATION; you may also need REFERENCE COUNTING if the buffers are shared between components.

Much of the main processing of the modem is COMPRESSION of the data sent on the phone line. Simple modem protocols may use SEQUENCE COMPRESSION; more complicated protocols will use TABLE COMPRESSION and ADAPTIVE COMPRESSION. To implement these more complicated protocols you'll require large and complicated data structures built up in RAM. To minimise the memory they use and improve performance, you can implement them with PACKED DATA and EMBEDDED POINTERS.

3. Large Web Server Project

Finally you might be working on a Java web server, which will provide an E-commerce Web and WAP interface to allow users to buy products or services. The server will connect to internal computers managing the pricing and stock control and to a database containing the details of individual users, via a local-area network.

RAM memory is relatively cheap compared with development costs, but there are physical limits to the amounts of RAM a server can support. The sever's operating system provides PAGING to increase the apparent memory available to the applications, but since most transactions take a relatively short time you won't want to have much of the memory paged out at any time. There will be many thousands of simultaneous users, so you can't afford simply to assign dozens of megabytes to each one.

You can ues SHARING so that all the users share just one, or perhaps just a few Java virtual machine instances. Where possible data will be READ ONLY, to make it easy to share. If the transaction with each user can involve arbitrarily complex data structures you can enforce a MEMORY LIMIT for each user. Maintainability and ease of programming are important so virtually all objects use VARIABLE ALLOCATION and GARBAGE COLLECTION.

The Internet connections to each user are a significant bottleneck, so you'll use ADAPTIVE COMPRESSION to send out the data, wherever the Web or WAP protocols support it. Finally, you may need to support different languages and page layouts for different users via RESOURCE FILES.

