Appendix: A Discussion of Forces

Version 04/12/99 17:23 - 37

How do you find the patterns relevant to your specific system?

If you're working on a project, the chances are you're already asking yourself "Which of the patterns in this book might I apply to my project?"

No serious software product ever had (or will have) as its mission purely to save memory. If it did, the solution would be simple: write no code! But all real software has other aims and other constraints. In the words of the patterns community [Coplien 1996, Vlissides 1998] you have other *forces* acting on you and your project. Each pattern provides a solution to a problem in the context of the forces acting on you as you make the decision.

A pattern's forces capture the problem's considerations and the pattern's consequences, to help you to decide when to use that pattern rather than another. Each pattern's initial problem statement identifies the major force driving that pattern, discusses of other forces affecting the solution. Then the pattern's Consequences section identifies how the pattern affects the configuration of the forces.

Some forces may be *resolved* by the pattern, that is, the pattern solves that aspect of the problem, and these forces form a pattern's positive benefits. Other forces may be *exposed* by the pattern, that is, applying the pattern causes additional problems, and these forces form a pattern's liabilities. You can then use further patterns to resolve the exposed forces, patterns that in their turn expose further forces, and so on.

This appendix answers the question above by asking in return "What other constraints and requirements do you have?", or,

What are your most important forces?

Identifying your forces can lead you to a set of patterns that you may – or may not – choose to use in your system.

Forces in this book

For all the patterns in this book the most important force is the software's memory requirements. But there are other important forces. The list below summarises most of the forces we've identified. In each case, a "yes" answer to the question generally means a benefit to the project.

The forces are in three categories:

- Non-functional requirements
- Architectural impact on the system
- Effect on the development process

The following tables give a brief summary of each force. The rest of this chapter examines each force in more detail, exploring the patterns that resolve and that expose each one.

Memory Requirements	Does the pattern reduce the overall memory use of the system?
Memory Predictability	Does the pattern make the memory requirements predictable? This is particularly important for real-time applications, where behaviour must be predictable.
Scalability	Does the pattern increase the range of memory sizes in which the program can

	function?
Usability	Does the pattern tend to make the easier for users to operate the system?
Time Performance	Does the pattern tend to improve the run-time speed of the system
Real-time Response	Does the pattern support fixed and predictable maximum response times?
Start-up Time	Does the pattern reduce the time between a request to start the system and its beginning to run?
Hardware and O/S Cost	Does the pattern reduce the hardware or operating system support required by the system?
Power Consumption	Does the pattern reduce the power consumption of the resulting system?
Security	Does the pattern make the system more secure against unauthorised access or viruses?

Table 1: Forces Expressing Non-functional Requirements

Memory waste	Does the pattern reduce the amount of memory in use but serving no purpose?
Fragmentation	Does the pattern reduce the amount of memory lost through fragmentation?
Local vs. Global	Does the pattern tend to help encapsulate different parts of the application, keeping them more independent of each other?

Table 2: Forces Expressing Architectural Impact

Programmer Effort	Does the pattern reduce the total programmer effort to produce a given system?
Programmer Discipline	Does the pattern remove restrictions on programming style, so that programmers can pay less attention to detail in some aspects of programming?
Maintainability and Design Quality	Does the pattern encourage better design quality? Will it be easier to make changes to the system later on?
Testing cost	Does the pattern reduce the total testing effort for a typical project?
Legal restrictions	Will implementing the pattern be free from legal restrictions or licensing costs?

Table 3: Forces Representing the Effect on the Development Process

The table in the back cover summarises a selection of the most important of these forces, illustrating how they apply to each of the patterns in the language. Each cell contains '☺' if the pattern normally has a beneficial effect in that respect (a "yes" answer to the question in the table above), '☹' if the pattern's effect is detrimental. A '😐' indicates that the pattern usually has an effect, but that whether positive or negative depends on circumstances.

The remainder of this chapter examines each force in more detail. For each force, we indicate the patterns that best resolve it, and the patterns that regrettably often expose it.

Forces related to non-functional requirements

The forces in this section concern the delivered system. How will it behave? Will it satisfy the clients' needs by being sufficiently reliable, fast, helpful and long-lived?

Memory Requirements

Does the pattern reduce the overall memory use of the system?

The single most important force in designing systems for limited memory is, unsurprisingly enough, the memory requirements of the resulting system – the amount of memory the system requires to do its job.

Patterns that resolve this force

- All the patterns in this book (Chapters N to M) resolve this force in one way or another.

Memory Predictability

Does the pattern make the memory requirements predictable?

Minimising a program's absolute memory requirements is all very well, but often it is more useful to know in advance whether a given program design can cope with its expected load, precisely what its maximum load will be, and whether it will exhaust the memory available to it. Often, increased memory requirements or reduced program capacity are better than random program crashes. In order to be able to determine that a program can support its intended load, or that it will not run out of memory and crash, you need to be able to audit the program to predict the amount of memory that it will require at runtime.

Predictability is particularly important for systems that must run unattended, where behaviour must be guaranteed and reliability is essential. In particular, life-critical systems must have predictable requirements. See, for example, the discussion in the **FIXED ALLOCATION** pattern.

Patterns that resolve this force

- **FIXED ALLOCATION** ensures your memory requirements do not change while the code is running. You can calculate memory requirements exactly during the design phase.
- **EMBEDDED POINTERS** allow you to calculate memory requirements for linked collections of objects easily.
- **PARTIAL FAILURE** permits pre-defined behaviour when memory runs out.
- A **MEMORY LIMIT** puts a constraint on the amount of memory used by any particular component.
- **DATA FILES** and **APPLICATION SWITCHING** handle only a certain amount of data at a time, potentially removing the chance of memory exhaustion.
- **EXHAUSTION TEST** verifies the system's behaviour on heap exhaustion.
- **CAPTAIN OATES** releases memory from lower priority tasks making it possible to have high priority tasks that complete reliably.

Patterns that expose this force

- **VARIABLE ALLOCATION** encourages a component to use unpredictable amounts of memory.
- **GARBAGE COLLECTION** makes it more difficult to determine in advance precisely when unused memory will be returned to the system.
- **COMPRESSION** (especially **ADAPTIVE COMPRESSION**) reduces the absolute memory requirements for storing the compressed data, but by an unpredictable amount.

- **COPY-ON-WRITE** obscures the amount of memory required for an object — memory only needs to be allocated when an object is first modified.
- **MULTIPLE REPRESENTATIONS** means that the amount of memory allocated to store an object can vary considerably, and in some uses, dynamically.

Scalability

Does the pattern increase the range of memory sizes in which the program can function?

Moore's Law [1997] states that hardware capacity increases exponentially, which means than the amount of memory available at any given cost decreases greatly over time. As a result, the amount of memory available tends to increase over time (or, rarely, the same devices can be sold more cheaply) [Smith 1999]. So long-lasting software needs to be *scalable* to take advantage of more memory if it is available.

Furthermore different users may have different amounts of money to spend, and different perceptions of the importance of performance and additional functionality. Such user choices require scalable software too.

Patterns that resolve this force

- **VARIABLE ALLOCATION** adjusts the memory allocated to a structure to fit the number of objects the structure actually contains, limited only by the available memory.
- **PAGING** and other **SECONDARY STORAGE** patterns allow a program access to more apparent RAM, by storing temporarily unneeded information on secondary storage. Adding more RAM improves the time performance of the system without affecting the functionality.
- **MULTIPLE REPRESENTATIONS** allows the system to size its objects according to the available memory.

Patterns that expose this force

- Designing a **SMALL ARCHITECTURE** requires you to make components responsible for their own memory use and accepting this responsibility can sometimes increase the complexity and decrease the performance of each component. The components bear these costs even when more memory become available.
- **FIXED ALLOCATION** (and **POOLED ALLOCATION** from a fixed sized pool) require you to commit to the size of a data structure early, often before the program is run or before the data structure is used.
- **SMALL DATA STRUCTURES**, especially **PACKED DATA**, trade time performance to reduce memory requirements. It can be hard to redesign data structures to increase performance if more memory is available.

Usability

Does the pattern tend to make the easier for users to operate the system?

Designing systems that use limited amounts of memory requires many compromises, and often these reduce the usability — ease of use, ease of learning, and user's speed, reliability and satisfaction — of the resulting system [Shneiderman 1997].

Usability is a complex, multifaceted concern, and we address it in this book only insofar as the system usability is directly affected by the memory constraints.

Patterns that resolve this force

- **PARTIAL FAILURE** ensures the system can continue to operate in low memory conditions.

- **CAPTAIN OATES** allows the system to continue to support users' most important tasks by sacrificing less important tasks.
- Other **ARCHITECTURAL PATTERNS** can make help make a system more consistent and reliable, and so more usable.
- Using **SMALL DATA STRUCTURES** can increase the amount of information a program can store and manipulate, to users' direct benefit.
- **PAGING** makes the system's memory appear limitless, so users do not need to be concerned about running out of memory.

Patterns that expose this force

- **SECONDARY STORAGE** patterns make users aware of different kinds of memory.
- **APPLICATION SWITCHING** makes users responsible for changing between separate 'applications', even though the may not see any reason for the separation of the system.
- **FIXED ALLOCATION** can make a system's memory capacity (or lack of it) directly and painfully obvious to the system's users.

Time Performance

Does the pattern tend to improve the run-time speed of the system?

Being small is not enough; your programs usually have to be fast as well. Even where execution speed isn't an absolute requirement, there'll always be someone, somewhere, who wants it faster.

Patterns that resolve this force

- **FIXED ALLOCATION** can assign fixed memory locations as the program is compiled, so they can be accessed quickly using absolute addressing.
- **MEMORY DISCARD** and **POOLED ALLOCATION** support fast allocation and de-allocation.
- **MULTIPLE REPRESENTATIONS** allows you to have memory-intensive implementation of some objects to give fast performance without incurring this overhead for every instance.
- **EMBEDDED POINTERS** can support fast traversal and update operations on link-based collections of objects.
- Most **GARBAGE COLLECTION** algorithms do not impose any overhead for memory management on pointer manipulations.

Patterns that expose this force

- **VARIABLE ALLOCATION** and deallocation cost processing time.
- **COMPRESSION**, especially **ADAPTIVE COMPRESSION**, requires processing time to convert objects from smaller compressed representations to larger computable representations.
- **COMPACTION** similarly requires processing time to move objects around in memory.
- Most **SECONDARY STORAGE** patterns, especially **PAGING**, uses slower secondary storage in place of faster primary storage.
- **REFERENCE COUNTING** requires up to two reference count manipulations for *every* pointer manipulation.
- **PACKED DATA** is typically slower to access than unpacked data.

- **SMALL INTERFACES** pass small amounts of data incrementally, which can be much slower than passing data in bulk using large buffer structures.
- **CAPTAIN OATES** can take time to shut down tasks or components.
- Indirect memory accesses via **HOOKS** can reduce the system's time performance.

Real-time Response

Does the pattern support fixed and predictable maximum response times?

Just as predictability of memory use — and the resulting stability, reliability, and confidence in a program's performance — can be as important or more important than the program's absolute memory requirements, so the predictability of a program's time performance can be more important than its absolute speed.

This is particularly important dealing with embedded systems and communications drivers, which may have real-world deadlines for their response to external stimuli.

Patterns that resolve this force

- **FIXED ALLOCATION , MEMORY DISCARD,** and **POOLED ALLOCATION** usually have a predictable worst case performance.
- **EMBEDDED POINTERS** can allow constant-time traversals between objects in linked data structures.
- **SMALL INTERFACES** ensure fixed amounts of data can be passed between components in fixed amounts of time
- **SEQUENCE COMPRESSION** can compress and decompress simple data streams in fixed amounts of time per item.
- **REFERENCE COUNTING** amortises memory management overheads at every pointer manipulation, and so does not require random pauses during a system's execution.

Patterns that expose this force

- **VARIABLE ALLOCATION** can require unpredictable amounts of time
- The time required by most **ADAPTIVE COMPRESSION** algorithms is dependent on the content of the information it is compressing.
- Some implementations of **MEMORY COMPACTION** may sporadically require a large amount of time to compact memory. If compaction is invoked whenever a standard allocator cannot allocate enough contiguous memory, then allocation will take varying amounts of time, and this performance will degrade as the free space decreases.
- Many **SECONDARY STORAGE** patterns take extra time (randomly) to access secondary storage devices.
- **COPY-ON-WRITE** requires time to make copies of objects being written to.

Start-up Time

Does the pattern reduce the time between a request to start the system and its beginning to run?

Start-up time is another force that is related to execution time, but clearly independent of both absolute performance and real-time response. For embedded systems, and even more crucially for PDAs and mobile phones, the time between pressing the 'on' switch and accomplishing useful work is vital to the usability and marketability of the system.

Patterns that resolve this force

- **PACKAGES** and **APPLICATION SWITCHING** allow a main module to load and start executing quickly; other modules load and execute later.
- **READ ONLY MEMORY** allows the CPU to access program code and resources immediately a program starts, without loading from secondary storage
- **SHARING** of executable code allows a new program to start up quickly if the code is already running elsewhere; **SHARING** of data reduces initial allocation times.
- **MEMORY DISCARD** can allocate objects quickly at the start of the program.
- **VARIABLE ALLOCATION** defers allocation of objects until they are needed.
- **COPY-ON-WRITE** avoids an initial need to allocate space and copy all objects that might possibly change; copying happens later as and when necessary.

Patterns that expose this force

- **FIXED ALLOCATION** and **POOLED ALLOCATION** require time to initialise objects or pools before the program begins running.
- **COMPRESSION** can require time to uncompress code and data before execution.
- Initialising from **DATA FILES** and **RESOURCE FILES** on **SECONDARY STORAGE** all takes time.

Hardware and Operating System Cost

Does the pattern reduce the hardware or operating system support required by the system?

Hardware or operating systems can provide facilities to directly support some of the patterns we have described here. Obviously, it makes sense to use these facilities when they are provided, if they address a need in your design. Without such support, you may be better off choosing an alternative pattern rather than expending the effort required emulating it yourself.

Patterns that expose this force

- **CAPTAIN OATES** needs a mechanism for individual tasks within a system to determine the system's global memory use, and ideally a means to signal memory-low conditions to all programs.
- **GARBAGE COLLECTION** is often provided in the virtual machines or interpreters for modern programming languages, or as libraries for languages like C++.
- **RESOURCE FILES** and **PACKAGES** need to load binary data such as executable files or font and icon files into running programs. This is easiest if implemented in the operating system.
- **PAGING** is much more efficient if it uses the page and segment tables of your processor, and in practice this requires operating system support.
- Similarly, **COPY-ON-WRITE** is implemented most efficiently if it can use hardware page table write protection faults.

Power Consumption

Does the pattern reduce the power consumption of the resulting system?

Battery-powered systems, such as hand-helds, palmtop computers and mobile phones, need to be very careful of their power consumption. You can reduce power consumption by avoiding polling, avoiding long computations, and by switching off power-consuming peripherals.

Patterns that resolve this force

- **READ-ONLY MEMORY** often requires no power to keep valid.

Patterns that exposethis force

- **SECONDARY STORAGE** devices, such as battery-backed RAM and disk drives, consume power when they are accessed.
- **COMPRESSION** algorithms need CPU power to compress and uncompress data.
- **PAGING** is particularly bad, since it can require secondary storage devices to be running continuously on battery power.

Security

Does the pattern make the system more secure against unauthorised access or viruses?

Security is increasingly important, with the advent of the Internet and private information being stored on insecure desktop or palmtop computers. As with forces like *memory predictablity* and *real-time response*, its generally not enough to claim that a system is secure, you also need to be able to audit the implementation of the system to see how it is built.

Patterns that resolve this force

- Information stored in **READ-ONLY MEMORY** cannot generally be changed so should remain sacrosanct.

Patterns that expose this force

- **SECONDARY STORAGE** devices, especially if used by **PAGING,** may store unsecured copies of sensitive information from main memory.
- **PACKAGES** can allow components of the system to be replaced or extended by other, insecure or hostile, versions.
- **HOOKS** allow nominally read-only code and data to be changed, allowing the introduction of viruses.

Architectural Impact

We can identify a different set of forces that affect the delivered system less directly – they're visible to the developers more than to the end-users.

Memory Waste

Does the pattern reduce the amount of memory in use but serving no purpose?

Some design approaches waste memory. For example low priority tasks may keep unnecessary caches; fully featured, large, objects may be allocated where smaller and more Spartan versions would do; and allocated objects may sit around performing no useful purpose.

Clearly it's generally good to avoid such wasted memory, even if in some cases it's worth accepting the penalty in return for other benefits.

Patterns that resolve this force

- **MULTIPLE REPRESENTATIONS** avoids unnecessarily memory-intensive instances of objects when a more limited representation will do the job.
- **SHARING** and **COPY-ON-WRITE** can prevent redundant copies of objects.
- **PACKED DATA** reduces the amount of memory required by data strucutres.

Patterns that expose this force

- **FIXED ALLOCATION** and **POOLED ALLOCATION** tend to leave unused objects allocated.

- Objects allocated by **VARIABLE ALLOCATION** can become memory leaks if they are no longer used and have not been explicit deleted.
- **REFERENCE COUNTING** and **GARBAGE COLLECTION** can also have memory leaks – objects that are no longer in use, but are still reachable from the system root.
- **MEMORY LIMITS** can waste memory by preventing components from using otherwise unallocated memory.
- **ADAPTIVE COMPRESSION** often needs to uncompress large portions of data into memory, even when much of it isn't required.

Fragmentation

Does the pattern reduce the amount of fragmentation?

Fragmentation causes memory to be unusable because of the behaviour of memory allocators, resulting in memory that is allocated but can *never* be used (internal fragmentation) or that has been freed but can *never* be reallocated (external fragmentation). See ***MEMORY ALLOCATION*** for a full discussion of fragmentation.

Patterns that resolve this force

- **MEMORY COMPACTION** moves allocated objects in memory to prevent external fragmentation.
- **FIXED ALLOCATION** and **POOLED ALLOCATION** avoid allocating variable-sized objects, also avoiding external fragmentation.
- **MEMORY DISCARD** avoids fragmentation – stack allocation has no fragmentation waste and discarding a heap discards the fragmentation along with it.
- **APPLICATION SWITCHING** can avoid fragmentation by discarding all the memory allocated by an application and starting over again.

Patterns that expose this force

- **VARIABLE ALLOCATION** supports dynamic allocation of variable sized objects, causing fragmentation dependent on your memory allocation algorithm.
- **FIXED ALLOCATION** and **POOLED ALLOCATION** generate internal fragmentation when they allocate variable-sized objects.

Local vs. Global Coupling

Does the pattern tend to help encapsulate different parts of the application, keeping them independent of each other?

Some programming concerns can be merely a local concern. For example stack memory is local to the method that allocates it. The amount of memory is determined directly by that method and affects only invocations of that method and any method called from it.

In contrast the amount of memory occupied by heap objects is a global concern. Methods can allocate many objects that exist after the method returns, and so the amount of memory allocated by such a method can affect the system globally. Some patterns can affect the balance between local and global concerns in a program, requiring local mechanisms to achieve global results or, on the other hand, imposing global costs to produce a local effect.

Patterns that resolve this force

- **SMALL ARCHITECTURE** and **SMALL INTERFACES** describe how program modules and their memory consumption can be kept strictly local.

- **PACKED DATA** and other **SMALL DATA STRUCTURES** can be applied to a local design for each structure, allowing redesign without affecting other components.
- **MULTIPLE REPRESENTATIONS** can change data structure representations dynamically, without affecting the rest of the program.
- **POOLED ALLOCATION** and **MEMORY LIMITS** can localise the effects of dynamic memory allocation to within a particular module.
- **MEMORY DISCARD** allows a set of local objects to be deleted simultaneously.
- **PAGING** allows most system code to ignore issues of secondary storage.
- **REFERENCE COUNTING** and **GARBAGE COLLECTION** allow decisions about deleting objects shared globally to also be made globally.

Patterns that expose this force

- **PARTIAL FAILURE** and **CAPTAIN OATES** require local support within programs to provide support for graceful degradation globally throughout the program.
- **VARIABLE ALLOCATION** shares memory between different components over time, so the local memory used by one component affects the global memory available for others.
- **SHARING** potentially introduces coupling between every client object sharing a given item.
- **EMBEDDED POINTERS** require local support within objects so that they can be members external (global) collections.

Development Process

The following forces concern the development process. How easy will it be to produce the system, to test it, to maintain it? Will you get management problems with individual motivation, with team co-ordination, or with the legal implications of using the techniques?

Programmer Effort

Does the pattern reduce the total programmer effort to produce a given system?

The cost of programmer time far exceeds the cost of processor time for all but the most expensive supercomputers (and for all except the cheapest programmers) – unless the software is very widely used. Some patterns tend to increase implementation effort, while others can reduce it.

Patterns that resolve this force

- **VARIABLE ALLOCATION** doesn't require you to predict memory requirements in advance.
- **GARBAGE COLLECTION** means that you don't have to worry about keeping track of object lifetimes.
- **HOOKS** allow you to customise code without having to rewrite it.
- **MEMORY DISCARD** makes it easy to deallocate objects.
- **PAGING** transparently uses secondary storage as extra memory.

Patterns that expose this force

- **PARTIAL FAILURE** and **CAPTAIN OATES** can require you to implement large amounts of checking and exception handling code.

- **COMPRESSION** patterns (especially **ADAPTIVE** Compression) may require you to implement compression algorithms or learn library interfaces.
- **COMPACTION** requires effort to implement data structures that can move in memory.
- Most **SECONDARY STORAGE** patterns require programmers to move objects explicitly between primary and secondary storage.
- **SMALL DATA STRUCTURES** can require you to reimplement parts of your program to optimise its memory use.
- **EMBEDDED POINTERS** can require you to rewrite common collection operations for every new collection of objects.

Programmer Discipline

Does the pattern remove restrictions on programming style, so that programmers can pay less attention to detail in some aspects of programming?

Some patterns depend upon you to pay constant attention to small points of detail, and carefully follow style rules and coding conventions. Following these rules requires a high level of concentration, and makes it more likely you will make mistakes. Of course once learned the rules do not greatly reduce your productivity, or increase the overall effort you will need to make.

Some patterns (like **PAGING**) reduce programmer discipline by using automatic mechanisms, but require effort to implement those mechanisms; others, like **REFERENCE COUNTING,** require discipline to use but do not take much effort to implement.

Patterns that resolve this force

- **GARBAGE COLLECTION** automatically determines which objects are no longer in use and so can be deleted, avoiding the need to track object ownership.
- **PAGING** uses secondary storage to increase the apparent size of main memory transparently, avoiding in many cases the discipline of **PARTIAL FAILURE**.
- **COPY-ON-WRITE** means that clients can safely modify an object regardless of whether it is **SHARED** or in **READ-ONLY MEMORY**.

Patterns that expose this force

- A **SMALL ARCHITECTURE** requires discipline to keep system and component wide policies about memory use, and to use **READ-ONLY MEMORY** and **RESOURCE FILES** as appropriate.
- **PARTIAL FAILURE** requires you to cater for memory exhaustion in almost all the code you write.
- **CAPTAIN OATES** may require you to implement 'good citizen' code that doesn't add to your current component's apparent functionality.
- **REFERENCE COUNTING** and **COMPACTION** may require you to use special handle objects to access objects indirectly.
- **POOLED ALLOCATION** and **MEMORY DISCARD** require careful attention to the correct allocation, use, and deallocation of objects, to avoid dangling pointers or memory leaks.
- You have to include **HOOKS** into the design and implementation of your components so that later users can customise each component to suit their requirements.

- Using **COMPRESSION** routinely (say for all string literals) makes programming languages literal facilities much harder to use.
- **EMBEDDED POINTERS** require care when objects can belong to multiple collections.

Design Quality and Maintainability

Does the pattern encourage better design quality? Will it be easier to make changes to the system later on?

Some design and programming techniques make it easier for later developers to read, understood, and subsequently change the system.

Patterns that resolve this force

- Taking the time to design a **SMALL ARCHITECTURE** and **SMALL DATA STRUCTURES** increases the quality of the resulting system.
- **HOOKS** allow a program's code to be extended or modified by end users or third parties, even if the code is stored in **READ ONLY MEMORY**.
- **PARTIAL FAILURE** supports other failure modes than memory exhaustion, such as network faults and disk errors.
- **SMALL INTERFACES** reduce coupling between program components.
- **MULTIPLE REPRESENTATIONS** allow objects implementations to change to suit the way they are used.
- **SHARING** reduces duplication between (and within) the components of a system.
- **RESOURCE FILES** and **PACKAGES** allow a program's resources — literal strings, error messages, screen designs, and even executable components — to change without affecting the program's code.
- **REFERENCE COUNTING**, **GARBAGE COLLECTION** and **PAGING** allow you to make global strategic decisions about deleting objects or using secondary storage.
- **APPLICATION SWITCHING** based on scripts can be very easily modified.

Patterns that expose this force

- **FIXED ALLOCATION'S** fixed structures can make it more difficult to change the volume of data that can be processed by the program.
- Code and data stored in **READ ONLY MEMORY** can be very difficult to change or maintain.
- **APPLICATION SWITCHING** can reduce a system's design quality when it forces you to split functionality into executables in arbitrary ways.
- **PACKED DATA** structures can be hard to port to different environments or machines.
- Collections based on **EMBEDDED POINTERS** are hard to reuse in different contexts.

Testing cost

Does the pattern reduce the total testing effort for a typical project?

It's not enough just to code up your program; you also have to make sure it works reliably (unless your product commands a monopoly in the market!). If you care about reliability, choose patterns that decrease the cost of testing the program, so that you can test more often and more thoroughly.

Patterns that resolve this force

- **FIXED ALLOCATIONS** are always the same size independent of program loading, so they always run out of capacity at the same time. This simplifies exhaustion testing.
- **READ-ONLY MEMORY** is easier to test because its contents are unable to change.
- **DATA FILES** and **RESOURCE FILES** help testing because you can use versions of the files to set up different test scenarios.

Patterns that expose this force

- **VARIABLE ALLOCATION, POOLED ALLOCATION, MEMORY DISCARD, MEMORY LIMIT,** and **MULTIPLE REPRESENTATIONS** require testing to check changes their in sizes and representations.
- **PARTIAL FAILURE** and **CAPTAIN OATES** have to be tested to check their behaviour both when memory is scarce, but also when it is abundant.
- **COMPRESSION** implementations should be tested to see that they perform in exactly the same way as implementations that don't use compression.
- Any kind of **SHARING** (including **HOOKS** and **COPY-ON-WRITE**) has to be exhaustively tested from the perspective of all clients of any shared objects, and also for any potential interactions implicitly communicated between clients via the shared object.

Legal restrictions

Will implementing the pattern be free from legal restrictions or licensing costs?

Some programming techniques are subject to legal restrictions such as copyrights and patents. Choosing to use these techniques may require you to pay license fees to the owner of the copyright or patent. Yet using third party software or well-known techniques is a crucial component of good practice in software development — indeed, making existing practices better known is the aim of this book.

Alternatively, some free software (aka Open Source) licences, notably the GNU General Public License, may require you to release some or all of your software with similar licence conditions. However the open source community is actively working to develop alternatives to proprietary techniques that can often be incorporated into all types of software without imposing onerous conditions onto the software development.

Patterns that expose this force

- **FILE COMPRESSION** algorithms from third parties are often subject to copyright or patent restrictions.
- **GARBAGE COLLECTION** and sophisticated **VARIABLE ALLOCATION** libraries usually come as proprietary software.

