Proceedings of the Memory Preservation Society

Patterns for managing limited memory

James Noble, Charles Weir

Introduction

The Memory Preservation Society is an old and august body — possibly one of the oldest computer groups in existence. Ours is a virtual society, for it exists solely in the minds of programmers. For our members are the programmers who find themselves challenged by the limitations of the real computers running their software, as compared with the ideal systems of their imagination. Their biggest problem is typically the dreaded shortage of memory. Over the years, our members have risen to this challenge with a variety of techniques, some traditional, others appropriate only to the object-oriented languages and sophisticated operating systems of today.

Of course, memory has become so much cheaper recently that for many environments the RAM memory use need not be a major constraint. However there are very few systems that can afford to ignore the issue entirely. Furthermore the recent development of palmtop computers and intelligent mobile phones has produced an entire new type of system where memory is limited. And users expect much more powerful software than they would ever have contemplated in the past.

Overview

In this paper we describe some of the most common patterns we've encountered in memory-challenged systems. We'll only consider limitations on RAM memory. Of course a typical system may have many other limitations: there may be constraints on graphics and output resources, disk space, network bandwidth, processing power, and many less obvious items. Some of the patterns, such as the optimisation ones, may apply as much to other constraints as to RAM constraints. Other patterns will not be so appropriate: compression, for example, may be unsuitable where the constraint is on processor power. Where appropriate, we've indicated in the individual patterns how they interact with other resource limitations.

The Patterns

The patterns follow a traditional pattern format with a name, problem description, solution ("Therefore"), forces ("Consequences"), known uses and further references. We've illustrated many with examples taken from a particularly memory-challenged system, the unique Strap-It-On™ wrist-mounted PC from the well-known company StrapItOn.

To make it easier to find specific patterns, we've grouped related patterns together into *Major Techniques*. For example, the major technique Compression includes many different forms of compression-based pattern. The following are the major techniques and the patterns associated with each one:

1 High-level and process patterns
- Think Small
- Memory Budget
- Memory Overdraft
- Make the User Worry
- Partial Failure
- Do the Decent Thing
- Test Small
- Memory Performance Assessment

2 Choose Suitable Data Structures
- Variable Sized Data Structure
- Fixed Size Data Structure
- Multiple Representation
- Sharing

3 Compression
- Byte Coding
- String Compression
- File Compression
- Short Names

4 Secondary Storage
- Program Chaining
- Data Chaining
- Autoloading
- Resource Files
- Segmentation
- Paging

5 Read-only Storage
- Copy-on-Write
- Hooks
- Pre-initialise data

The Pattern Structure

In writing this paper, we been surprised at the large number of memory-saving patterns we've found. To prevent this material leading to an unmanageably large paper, we've used a more condensed pattern structure than most pattern languages. As in Christopher Alexander's pattern format [Alex 77], each pattern highlights and discusses a problem, then concludes:

Therefore:

And it follows this with a specific recommendation and a discussion how to implement this recommendation. Further sections then describe *Consequences*, *Known Uses*, and related patterns and literature ('*See Also'*). Throughout the pattern descriptions we refer to other patterns using **pattern names in bold.**

Perhaps the most vital discussion for a pattern is of the *forces*, or considerations and consequences to help readers to decide when to use that pattern rather than another. In each pattern, we've identified in the problem statement at least one major force which drives each pattern - in addition to the need to restrict memory use. Then in the *consequences* section, we identify other the other implications, both positive and negative. To distinguish the positive from the negative consequences of each pattern, we describe the positive consequences first, the negative ones second, and partition the two with the word 'However:'.

Pattern Organisation

The list below shows some of the major forces we've identified in the patterns. In each case, a "yes" answer to the question is generally good. We've described the forces in more detail in the specific patterns.

Memory Requirements	Does the pattern reduce the overall memory use of the system?
Predictability	Does the pattern make the memory requirements predictable? This is particularly suitable for real-time applications, where behaviour must be predictable.
ROM vs. RAM	Does the pattern tend to encourage the use of cheaper Read-Only memory in preference to RAM?
Secondary Storage vs. RAM	Does the pattern tend to shift memory use towards cheaper secondary storage in preference to typically more limited RAM?
Time Performance	Does the pattern tend to improve the run-time speed of the system?
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 have to pay less attention to detail in all aspects of programming?
Testing cost	Does the pattern reduce the total testing effort for the application development?
Hardware and O/S Cost	Does the pattern reduce hardware or operating system costs?
Local vs. Global	Does the pattern tend to help encapsulate different parts of the application, keeping them more independent of each other?
Design Quality	Does the pattern encourage better design quality? Will it be easier to make changes to the system later on?
Usability	Will the pattern tend to make the easier for users to operate the system?
Legal restrictions	Will implementing the pattern be free from legal restrictions or problems?

Table 1: The major forces we've found in these patterns

Table 2 below shows how these forces 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.

	Memory Reqs	Predictability	ROM vs RAM	Sec. Storage vs. RAM	Time Performance	Progr. Effort	Progr. Discipline	Testing cost	HW and OS Cost	Local vs Global	Design Quality	Usability	Legal restrictions
Think Small	☺					☹	☹			☹			
Memory Budget	☺	☺				☹	☹			☺	☺		
Memory Overdraft		☺				☹	☹			☹			
Make the user worry	☺	☺				☺		☺				☹	
Partial Failure	☺	☺				☹	☹	☹			☹	☺	
Doing the Decent Thing	☺	☺				☹	☹	☹	☹	☹		☺	
Test Small						☹		😐				☺	
Memory Assessment	☺					☹							
Choose DS	☺	😐			😐			😐			☺		
Variable Sized DS	☺	☹			☹	☺				☹	☺		
Fixed Size DS		☺			☺	☹				☺		☹	
Multiple Representation	☺				☺	☹		☹			☺		
Dynamic Multiple	☺				☺	☹		☹			☺		
Compression	☺				☹	☹	☹						
Sharing	☺		☺			☹		☹					
Byte Coding	☺				😐	☺	☹	☹					
String Compression	☺				😐	😐		😐					
File Compression	☺		☹		☹	☹		☹					☹
Short Names	😐				☺	😐	☹	😐			☹		
Secondary Storage	☺				😐	😐	😐		😐	😐		😐	
Program Chaining	☺			☺	☹	😐					☹		
Data Chaining	☺	☺		☺	☹	☹					☹	☹	
Autoloading	☺			☺		☹			😐				
Resource Files	☺			☺	☹	☹	☹	☹	😐		☺	☺	
Segmentation	☺			☺	☹	☹	☹		☹				
Paging	☺	☺		☺	☹	☺	☺		☹	☹	☺		
Read-only Storage	☺		☺		☹	☹							
Copy-on-write	☺		☺		😐	☺			😐	☹			
Hooks	☺		☺		☹	☹	☹	☹			☺		
Pre-initialise data			☺	☺	☺	☺	☺						

Table 2: Effects of the Major Forces on each Pattern

Major Technique: High-level and Process Patterns

Developing small systems is as much a state of mind as a series of techniques. No one technique will restrict memory use; programmers must be working continually throughout the project to keep memory requirements down. So to start with, let's examine some patterns that encourage this state of mind in a project.

Think Small Pattern

How should you approach a small system?

You have to program a small system — that is, a system with *memory requirements* which will be hard to meet. The program size will be the total of the *sizes* of all the modules resident at a time, so every component of the system will add to the total size. Programmers (especially those without small system experience) *lack discipline* and will be sloppy about limiting their memory use — after all, in typical environments, it's bad engineering to spend *programmer effort* unnecessarily, and it's always more fun to add another feature than to reduce memory use. For example, early versions of the Strap-It-On™ Wrist Mounted PC came with fifteen different Wrist Accessory packages, but only one could run in the available memory.

Therefore: *Imagine the system is smaller than it is!*

Encourage every team member to keep a very tight control on the memory use. Develop a culture where memory saving is a habit. Use design and code reviews to exorcise wasteful features, habits and techniques.

There are many things to look out for, both during specification and coding:

- Avoid unnecessary data structures.
- Choose compact data structures (e.g. atoms and enumerated types rather than strings, bytes rather than machine words, etc.).
- Avoid language features that cost memory (e.g. C++ Vtbls, exceptions).
- Reduce the quality of images and sound.
- Reduce the list of unnecessary 'dot point' features (that is, features that are peripheral to the program's main purpose), such as desk calculators, spelling checkers and graphical user interfaces.
- Reduce hidden features, such as debugging information, self-checking code and 'gang screens' ('splash screens').

Consequences

You end up with a small system, which meets its *memory requirements*. **However:** you have to worry about this all the way through the development, so you need more *programmer effort* and *programmer discipline*, and programmers may trade off *local vs. global* memory use.

Known Uses

Most development teams for small machines use these patterns to some extent. For example the Microsoft CE versions of MS Windows applications cut down enormously on the features of the applications provided. Many developments support two compilation modes, for debug and for release; the debug mode includes processing and data attributes not present in the release mode. The Psion Series 5 development used 4-tone bitmaps in preference to any other, culled the dot-point features and enforced strict development policies to allow a large number of applications to run simultaneously in a small memory area.

See Also

There are two alternative strategies for thinking small. If you're a pessimist, prefer to think ahead, or are facing particularly tight *memory requirements*, then your should prepare a **Memory Budget** so that you can plan your use of memory. If you're an optimist, prefer to fix problems only when they actually arise, or your *memory requirements* are not particularly tight, you should consider performing a **Memory Performance Assessment** and dealing with memory limitations once the program is running. In either case, you need to **Choose Suitable Data Structures** and use **Compression** to reduce the *memory requirements,* or move portions of your program into **Secondary Storage** or **Read-only Storage**. You may need to **Make the User Worry** about memory, **Test Small** to be sure you support graceful degradation in the face of **Partial Failures**, and, where necessary, make programmers **Do the Decent Thing**, relinquishing memory to give other programs a chance to run.

Memory Budget Pattern

How do you keep control of memory in a very tight project?

Your project has very tight *memory requirements*. Unless memory constraints are a high priority it's unlikely that the system will work correctly.

Therefore: *Draw up a memory budget, and worry about it a lot. Define targets for the whole system and for each component as part of the specification process.*

The most important thing about a memory budget is that people working on the project take it seriously. Some environments provide memory use monitors or resource limits which can be used to enforce memory budgets, and these can increase the seriousness of the budgets.

Consequences

Having specific targets for memory use greatly increases the *predictability* of the memory use of the resulting program. Because developers face specific targets, they can make decisions *locally* where there are trade-offs between memory use and other constraints. It's easy to identify problem areas, and to see which modules are keeping their requirements reasonable, so a budget increases *programmer discipline*. **However:** defining, assigning and managing the budgets requires significant *programmer effort*. Developers may achieve their *local* budgets in ways that have unwanted *global* side effects such as poor *time performance*, off-loading functionality to other modules or breaking necessary encapsulation (see [Brooks75]). Runtime enforcement of memory budget requires *hardware or operating system support.*

Known Uses

The OS/360 project used memory budgets [Brooks75]. The Psion EPOC32 system implements a de-facto memory budget by limiting the heap space of each application in the development environment. Even more constraining, many environments allow you to specify hard run-time limits, e.g. IBM UNIX and NewMonics' PERC Real-time Java [Nilson95].

See Also

Memory Performance Assessment provides an alternative solution to the same problem. **Memory Overdraft** arrangements can help when your Memory Budget goes down the gurgler. For some kinds of programs you cannot produce a complete budget in advance, so you may need to allocate memory coarsely between the user and the system, and then **Make the User Worry** about memory.

[Gilb88] describes techniques for 'attribute specification' appropriate for defining the project's targets.

Memory Overdraft Pattern

What happens when your memory budget goes wrong?

You've produced a memory budget, but *memory requirements* are *unpredictable*, and hard to estimate accurately. As your development proceeds, it becomes clear that some components will need more than their budgeted amount of memory — either temporarily, to handle infrequent special conditions, or permanently.

Therefore: *Include overdraft provisions in your memory budget.*

Ensure your memory budget has some slack. When a component needs more memory, and you can't reduce its size in any other way, you can grant it more memory from the overdraft. Either the remains of this overdraft, or a separate dynamic overdraft memory allocation, can supply extra memory in emergencies at runtime.

Consequences

Your budget will be more resilient in the face of development realities, increasing the overall *predictability* of the program's memory use. **However:** Managing an overdraft takes *programmer effort*. Knowledge about the overdraft may encourage programmers to reduce their *discipline* and take it for granted, reducing the integrity of the budget. An overdraft is typically shared by all the modules in a program, so programmers can use an overdraft to get extra *local* memory while reducing the amount of extra *global* memory available.

Known Uses

The OS/360 project included overdrafts as part of their budgets [Brooks75]. Prograph on the Macintosh includes a 'Rainy Day' fund of memory which programmers can call on in an emergency [MacNeil 95].

See Also

A Memory Overdraft may be exhausted at runtime, so **Test Small** to ensure **Partial Failures** when the overdraft is overdrawn. You can often **Make the User Worry** about memory as an alternative to providing a Memory Overdraft. An inactive part of the program should **Do the Decent Thing** and release some memory if an active part of the program is making heavy use of the Memory Overdraft.

Make the User Worry Pattern

Sometimes the user is in the best position to allocate or manage the use of memory.

In many cases, especially in interactive systems, *memory requirements* cannot really be *predicted* in advance. This is because the uses for the memory will depend critically on what the user chooses to do with the system. If you try to produce a generic budget, you will over-allocate the memory requirements for some parts of the program, and consequently have to under-allocate others. If you allocate a large amount of space for a **Memory Overdraft,** you effectively postpone memory planning until runtime, and are consequently more likely to run out of memory**.** For example, the memory requirements for a user typing small memos on the Word-O-Matic on the Strap-It-On PC will differ greatly from a user writing a PhD thesis — one may require a large memory allocation for clip-art, and the other a large allocation for actual text.

Therefore: *Make the system's memory model explicit in its user interface, so the user can worry about memory.*

On the basis of your **Memory Budget**, allocate memory coarsely between the user and the program. Use a **Memory Budget** and other patterns from this paper to manage the systems memory. Design a model of the way user memory will be used — either as a **Fixed Size Data Structure,** with a fixed number of locations that storing data of a particular type, or as a **Variable Size Data Structure** which can contain any number of data of varying types. Expose this 'memory model' in your program's user interface, and let the user manage the allocation of user memory directly.

There are number of techniques which can expose a memory model to its users:

- Constantly display the amount of free memory in the system.
- Provide tools, which allow users to query the contents of their memory, and the amount of memory remaining.

- Generate warning messages or dialogue boxes as the system runs out of memory.
- Make the user choose what data to overwrite or delete when they need more memory.

For example, Word-O-Matic allows the user to choose how much memory should be allocated to image caches, and uses all the otherwise unallocated space to store the document. Word-O-Matic also displays the available memory to the user.

Consequences

The system can deliver more behaviour to the user than if it had to make pessimistic assumptions about its use of memory. The user can adjust their use of the system to make the most of the available memory, reducing the *memory requirements* for performing any particular task. Although the way user memory will be allocated at runtime is unpredictable, it is quarantined within the **Memory Budget**, so the memory use of the system as a whole is *more predictable*. **However:** the user now has to worry about memory, whether they want to or not, so the system is *less usable*. Given a choice, users will choose systems where they do not have to worry about memory. You have to spend *programmer effort* designing the user's memory model, and making the memory model visible to the user.

Known Uses

The Macintosh and MS-Windows systems display the amount of free disk space in every desktop disk window. Application like MS Windows' Notepad have fixed limits on the size of their application data. The Acorn Archimedes can present a bar graph showing the memory use in the system, and the user can drag the bars to change memory allocations. Smalltalk-80 produces a sequence of notifier and warning messages as it was running out of memory. Music synthesisers and programmable calculators often provide a small fixed number of memory locations for programs, and users think of these systems have having just that number of memories. The Ensoniq Mirage is the ultimate example of this technique — the user must allocate memory for sound sample storage by entering two digit hexadecimal digits using only increment and decrement buttons.

See Also

The memory model exposed to the user may be a **Fixed Size Data Structure** or a **Variable Sized Data Structure**. **Functionality a la Carte** [Adams 95] can be used to present the costs and benefits of memory allocations to the user. A basic **Memory Budget** and **Memory Overdraft** can provide an alternative which does not require the user to manage memory explicitly, but which will have higher *memory requirements* to provide a given amount of functionality.

Partial Failure Pattern

Also known as: Graceful degradation

How do we deal with unlimited demands for memory?

No matter how much we do to reduce the program's *memory requirements*, there will still be situations where the program runs out of resources.

Therefore: *Allow the program to fail partially, or have its behaviour degrade but not stop.*

When a program runs out of memory, it doesn't just stop but continues as best it can. This usually means aborting the operation that used up all the memory. It's important that this frees up enough memory to allow further operations. Alternatively, and usually better, the program may continue but in a degraded mode. For example when the Word-O-Matic fails to allocate the memory required for the voice output of a document, it continues to provide a screen-only display. More typical examples of degraded modes are as follows:

- Loading a font may fail; in this case we can use a standard system font.
- Displaying images may fail; we can leave them blank or display a message.
- A detailed calculation may fail; we can use an approximation.

- A compression process may fail; we might store the uncompressed data.

Often objects encapsulate this failure to hide it from their clients. The Word-O-Matic's voice output module still accepts commands from its clients even in its 'out of memory' state, which makes programming its clients much simpler.

Consequences

This approach significantly improves the program's *usability*. With careful design, even the degraded modes can provide essential functionality. By ensuring the program can continue to operate within a given amount of memory, partial failure decreases the program's minimum *memory requirements* and increases the *predictability* of those requirements. **However:** it is hard work to program, requiring *programmer discipline* to apply consistently and *programmer effort* to actually implement, because it requires *local* mechanisms to provide a *global* effect. Supporting partial failure significantly increases the complexity of each module, increasing the *testing cost* because you must try to test all the failure modes.

Known Uses

If Netscape fails to load a font due to insufficient memory, it continues with standard fonts. Microsoft Powerpoint will use standard fonts and omit images if it runs out of memory. In UNIX, if there's insufficient memory to run a job, that job fails rather than the whole system dying.

See Also

It's essential that this new behaviour be properly tested, as described in the **Test Small** pattern. Whereas the Partial Failure pattern describes what a program should do when it runs out of memory**,** **Do The Decent Thing** describes what a program should do when another program in the system runs out of memory.

Do the Decent Thing Pattern

How do we prioritise demands for memory?

A program's *memory requirements* are not equally important. At any given time, some of a program's *memory requirements* will be more important to its clients or users than others. For example, when someone is using the Strap-It-On PC's word processor to edit document, the user doesn't care what the fractal screen background looks like. Some programs may be exceeding their share of the **Memory Budget**, by holding on to a **Memory Overdraft** at runtime. You can increase a system's *usability* by spending scarce resources doing what the user actually wants.

Therefore: *When a system runs out of memory, surrender memory used by less vital components rather than fail the most important tasks.*

When the system's memory is low, indicate this condition to the running programs. When a program receives this signal it should release its inessential memory, or in more extreme situations, close down. Often programs use memory to cache information which is not strictly required, in order to improve time performance. This cached information prevents the resources from being available to other processes which may need it for more vital operations. If there is no support for signalling memory conditions, an alternative is for programs to keep track of the free memory situation by regular polling, and free inessential resources (or close down) when memory becomes short. For example when the Word-O-Matic runs out of memory its background 'Fizzy™' fractal generator automatically closes down. As a result, the word processor's *memory requirements* can be met.

Consequences

This approach increases the systems *usability*, and by allocating memory where it is most needed, reduces the system's *memory requirements.* Programs releasing their overdrawn memory also increase the *predictability* of the system's memory use. **However:** Doing the decent thing requires *programmer discipline* to consider voluntarily releasing resources, and *programmer effort* to implement. Releasing resources can reduce the program's *time performance*. Programs need to be *tested* to see that they do release resources, and that they continue to perform successfully afterwards. Because all programs must handle the memory low signal, doing the decent thing is easier with *operating system support*. This is also an example of *global mechanism* that introduces *local complexity* to handle the signal.

Known Uses

Windows has an 'about to run out of memory' signal, to which well-written Windows applications will respond. For example networking 'IP stacks' may release some of the cached IP address map; web browsers may release cached pages; or a word-processor might delete 'undo' information. In Psion's EPOC32, each memory heap caches redundant blocks in case the process needs them again. When memory runs out, the system frees these blocks and retries the operation. The Apple Macintosh supports unlocked memory blocks which the OS may reclaim when memory is short. These are often used for read-only caches of disk data.

See Also

Doing the Decent Thing allows programs to use **Memory Overdrafts** without affecting the overall integrity of the **Memory Budget.** Where Do The Decent Thing describes what a program should do when another program in the system runs out of memory, **Partial Failure** describes what a program should do when it runs out of memory.

Test Small Pattern

Also known as: Graceful Degradation Testing, Out-of-memory testing.

How do you ensure that your programs work correctly in out of memory conditions?

Programs that deal gracefully with resource failures — say by using the **Memory Overdraft** or **Partial Failure** patterns — have a large number of extra situations to deal with, because each resource failure is a different execution event. Ideally, the program should be tested in every situation which will arise in execution, but exhaustive testing will greatly increase the *testing cost*. Testing for memory exhaustion failures is particularly difficult because these events are by definition exceptional, so they will occur mostly when the system is heavily loaded or has been executing for a long period.

Therefore: *Use testing techniques which simulate memory exhaustion.*

Use a version of the memory allocator that fails after a given number of allocations, and verify that the program behaves sanely for values of this number. Also use another version of the allocator that inserts random failures. Verify that the program implements **Partial Failure** by taking alternative steps to get the job done, or **Makes the User Worry** by reporting to the user if this is not possible.

You should combine partial failure testing with more traditional memory testing techniques such as the use of conservative garbage collectors to verify that the program does not cause resource leaks.

Consequences

Using specialised testing techniques reduces the *testing cost* for a given amount of trust in the program. **However:** being reasonably certain that the resulting program will work correctly will still have a large *testing cost*. This approach also needs *programmer effort* to build the specialised memory allocators to support the tests. Testing doesn't always detect the results of random and time-dependent behaviour, for example where two threads are both allocating independently.

Known Uses

Many development environments provide tools or libraries to allow memory failure testing. The Purify environment for C++ supports random and predictable failure of C++'s memory allocator. Psion's EPOC32 provides debug versions of the allocator with similar features and these are used in module testing of all the EPOC32 applications.

Memory Performance Assessment Pattern

How do you stop memory constraints dominating the design process to the detriment of other requirements?

Your system has to meet particular *memory requirements*, but other requirements are more important. Implementing and tracking a Memory Budget costs *programmer discipline* and *programmer effort* that should be better directed towards other requirements. For example, if you're developing a new operating system release for desktop PCs, then integrating a web browser into the desktop, providing AI-based help systems and shipping the system less than a year late, will all be more important than the amount of memory the system occupies.

Therefore: *Implement the system, paying attention to memory requirements only where these have a significant effect on the design. Once the system is working, identify the most wasteful areas and optimise their memory use.*

Using this approach, development proceeds much as per usual, with no special attention to memory use. This is fine, as long as the resulting program meets its memory requirements. If (as often happens) the program does not meet its *memory requirements* you have to perform memory optimisation.

Consequences

The team develops the system effectively and faster, because they are not making unnecessary optimisations — a given amount of programmer *effort* gets you more software with better *time performance* and higher *design quality*. **However:** The *memory requirements* of the resulting program will be hard to *predict.* It's more difficult to leave memory optimisation to last than performance optimisation, where you can concentrate on algorithmic hot-spots, because memory optimisation requires changes to data structures which tends to affect large amounts of code. So memory optimisation can require a large amount of *programmer effort*, reduce *design quality*, require *local vs. global* tradeoffs, and increase the *cost of testing.*

Known Uses

This pattern occurs very frequently in projects, since it is what happens by default. Typically a team implements a system, then discovers it uses too much memory. Memory optimisation follows.

See also

Most of the following patterns in the language optimise a program's memory use. The patterns **Lazy Optimisation** and **Optimise the Right Place** in [Auer+95] address speed optimisation, but the techniques apply equally to space optimisation. Two relevant low-level patterns in the same language are **Transient Reduction** and **Object Transformation**.

If you are facing particularly tight *memory requirements,* prefer to think ahead or are a pessimist, then you should prepare a **Memory Budget** in advance, so you can plan your use of memory.

Major Technique: Choose Suitable Data Structures

Where *memory is limited*, we must be particularly careful to choose appropriate data structures to minimise the program's *memory requirements*. The data structures and algorithms which are appropriate where memory is unlimited may be far too prodigal where memory is restricted. For example a typical implementation of an address database might store copies of information for indexes as well as the actual data. Porting such an approach to the Strap-It-On™ PC would restrict the database size so as to make it virtually unusable.

Therefore: *Be careful to choose data structures to optimise memory use.*

Analyse the program's data to work out precisely what information the program needs to store, and to determine its characteristics — how much data the program must deal with, whether this varies over the course of a single program run across different runs, whether it consists of a few large objects or many small objects, and so on. Design data structures to hold the data, and evaluate their memory requirements against your memory budget. When you're under budget, you're done. For example, the Strap-It-On™ PC's budget has enough space to store the address records but not the indexes, so its address program uses a sorted data structure which does not need extra space for an index.

Consequences

Choosing suitable data structures can reduce *memory requirements*, and the time spent can increase the *quality of the program's design.* **However:** memory budgeting and data structure design takes *programmer effort* to do well. The *predictability* of the program's memory use, the *testing costs*, and the program's *time performance* may or may not be affected, depending upon the chosen structures.

See Also

The following patterns explore several options for choosing data structures**. Variable Sized Data Structure** and **Fixed Size Data Structure** are fundamental opposing choices for data storage. A further pattern, **Multiple Representation**, and its variants (**Dynamic Multiple Representations** and **Basic Type with Object Wrapper**) explore using the encapsulation facilities of O-O languages to hide further memory optimisations from users of a specific class.

In many cases, good data structure design alone is insufficient to manage your program's memory. The **Compression** patterns, and the **Sharing** pattern in particular, describe how memory requirements can be reduced by explicitly spending processor time. The demand for writable primary storage can be reduced further by moving less important data into **Secondary Storage** and constant data into **Read-only Storage.**

Variable Sized Data Structure Pattern

How can you avoid unused empty space in a data structure?

You have a *variable amount of data to store*, and you're prepared to handle the situation where there's not enough memory available to meet your *memory requirements*. You could use a **Fixed Sized Data Structure**, but a large amount of memory in the structure would be wasted. For example, the Strap-It-On™'s famous Word-O-Matic word-processor stores the part of the user document currently displayed, in RAM. This is an unpredictable size, depending on the current screen size, and the fonts and paragraph formats selected by the user. And since there's also a voice output feature, it saves the vocal emotions for each paragraph!

Therefore*: use a variable sized data structure*

Split the data into different data structures, and allocate and free them dynamically as required. Implement the structure using dynamic structures such as linked lists, variable-length arrays and trees. Typically these will use references rather than embedded data, so you may need to use reference counting or garbage collecting to recover memory from linked structures. For example, Word-O-Matic dynamically requests memory from the system to store the user's documents, so to produce voice output Word-O-Matic just requests more memory. If the program runs out of memory, Word-O-Matic just tells the user to close another application.

Consequences

Variable-size structures avoid unused empty space, thus *reducing memory requirements* overall and generally *increasing design quality.* **However:** There will be a *memory overhead* to manage the size information (counts and pointers), and a *time overhead* to manage memory allocation and deallocation, which may be *global* to the language runtime system, the operating system (or in the case of the ill-fated Intel 432, in hardware). It may also become *hard to predict* the memory required for typical and worse case scenarios. Furthermore, you must be prepared to handle the situation where memory runs out during the construction of the structure, *adding additional complexity* to the code and requiring additional *programmer effort* to manage.

Known Uses

Virtually all O-O programming languages support this pattern by encouraging the use of dynamically allocated memory and by providing dynamic collections (Vectors, Sets, etc.). Therefore the vast majority of C++, Smalltalk, and Java applications use this pattern. Other languages which encourage dynamic memory allocation also encourage this pattern; hence most C, Pascal, and Lisp programs use this pattern too. Most environments provide dynamically-allocated strings, which use variable-length data structures, and so-called "dynamic languages" like Smalltalk and Java provide built in garbage collectors to manage dynamically varying storage requirements.

See also

Using a **Fixed Size Data Structure** avoids the overhead, unpredictability and complexity of a variable sized structure at the cost of often allocating more memory than is actually required. **Multiple Representations** (or **Dynamic Multiple Representations**) of data, and **Basic Types with Object Wrappers** can provide variable-sized structures for particular cases.

The **Hypoth-a-Size** collection pattern optimises allocation of variable-sized structures [Auer+95]. There are many books describing variable sized data structures and their use, for example [Aho+82].

Fixed Size Data Structure Pattern

How can you ensure you will always have enough memory?

With some operations it's vitally important that the programs *memory requirements* don't exceed the memory available – that you don't run out of memory during processing. For example the Strap-It-On's Muon-based satellite communication system must be prepared to accept short messages from extra-terrestrials at any time; it guarantees to store the last five messages. Running out of memory while receiving a message would not be acceptable.

Therefore: *use a fixed size data structure*

Allocate a fixed amount of memory to store the data structure, based on the analysis you carried out to produce your **Memory Budget**. Keep it unused until it's needed, or fill it, and then reuse blocks as required. To provide memory for a larger number of small objects, allocate space for a fixed-size number of items from a fixed-sized pool. Typical data structures for this are static buffers, arrays, circular buffers and circular lists. For example, the Strap-It-On's MuonTalk driver allocates space to store five 1024-byte messages when it starts up, and keeps this space as long as the Strap-It-On is running.

Consequences

The fixed size means you can *predict the system's memory use* exactly, and the *time required* to initialise memory is constant. These two features make this pattern particularly suitable for real-time applications. In addition, there is *no space overhead* for keeping counts or using pointers, and *no global overhead* for a garbage collector. Using **Fixed Size Data Structures** also *reduces programmer* *effort*, makes programs *easier to test* (since they either have enough memory or they don't), and often makes programs *more reliable* as there is less to go wrong. **However:** to handle the worst case you will probably budget and allocate more memory than necessary on average, which will increase the program's *memory requirements*, particularly in systems with many concurrent applications. Nowadays programs which use fixed size structures are often seen as *lower-quality* designs, although this probably says more about fashion than function!

Known Uses

Many traditional procedural languages do not support dynamic memory allocation directly. So for example most FORTRAN and COBOL programs will use this pattern. Many real-time systems use this pattern too: dynamic memory allocation (or rather memory compaction) takes an unpredictable time and real-time systems must respond within fixed, defined, times. This pattern is also used frequently in safety-critical systems, since some dynamic memory allocation systems may crash if they run out of memory. Strings based on static buffers use this pattern too.

See Also

Variable Sized Data Structures save memory by allocating only enough memory to meet immediate requirements, but require more effort and overhead to manage and make memory requirements harder to predict.

Multiple Representations Pattern

How can you support several different implementations of a structure?

There is no one "best" representation for a data structure; often, there is not even just one appropriate representation. Each representation has different *memory requirements.* For example, in the Strap-It-On™ word-processor, a word may be represented as a series of characters, a bitmap, or a string of phonemes. Depending on the current output mechanism (a file, the screen, or the vocaliser) each of these representations might be appropriate, making the other representations inappropriate in that case.

Therefore: *Have several different implementations of a derived class, and have the programmer choose the most appropriate for each specific task.*

Design a common abstract interface for all the structures, which does not depend on the particular representation. Then implement that interface appropriately for each implementation. Reference each implementation via an **Abstract Class** [Woolf97] or an **Adapter** class [Gamma+94] so that users don't have to be aware of the implementation.

Consequences

The system will use the most appropriate implementation for any task, *reducing processing time* which would be necessary to use an inappropriate representation. Code using each instance will use the common interface, and need not know the implementation, *reducing programmer effort* on the client side, increasing *design quality* and *reusability*. Representations can be chosen *locally* for each data structure. **However:** the resulting duplication can *increase total memory requirements*, *increase programmer effort* on the server side because multiple implementations are more complex than a single implementation, *increase testing costs* overall, because each alternative implementation must be tested, and *require programmer discipline* to avoid unnecessary dependencies by clients on a particular implementation.

Known Uses

Psion Software's EPOC32 C++ environment handles strings using 'Descriptors'. These provide many different representations of strings: in ROM, in a fixed-length buffer and in a variable length buffer. Users of the strings see only two base classes: one for a read-only string, the other for a writable string. Smalltalk's collection classes also use this pattern: all satisfy the same protocol, so a user need not be aware of the particular implementation used for a given collection. Java's collection classes have a similar feature: although the classes have no common interface, all support the Enumeration interface, which acts as an **Adapter**.

See Also

Dynamic Multiple Representations avoid the space cost by storing only one representation, and converting between representations dynamically, depending upon they way the data is used. **Basic Types with Object Wrappers** dynamically change representations to use first-class objects only when the full generality of objects is worth the cost.

Variant: Dynamic Multiple Representation Pattern

How can you handle a data structure that can change dynamically between a wide range of sizes?

Suppose you have some data which can change radically, particularly in its *memory requirements*. There are various possible representations for many kinds of data, but any one representation often suits a particular volume of data. For example, Word-O-Matic could be used to edit anything from a small memo to large document. Formatting, graphics, and vocal inflexion can be stored within the document or regenerated upon demand, trading off memory requirements for processing time and responsiveness.

Therefore: *Support multiple representations of a data structure, and use the most appropriate, dynamically changing between representations at run-time.*

Make each implementation of the **Multiple Representations** be able to convert itself to each other implementation — using the **Bridge** pattern to keep the interface and identity of the object constant when its internal structure changes. Determine the conditions when each representation is appropriate — these conditions may be internal to the data structure (the volume of data it contains) or external to it (the amount of memory in the system). Change the representation when the conditions change.

Consequences

This approach makes the best use of available resources by *reducing memory requirements*, and can often be implemented *locally* for each data structure. **However:** it requires *space and time to manage* changing between representations. It also means *more complexity* in the code, and *more complicated testing* strategies (since all implementations and transitions must be tested), increasing *programmer effort* and making memory use *harder to predict*. These days this kind of complexity is often seen as a sign of *high-quality* design because subsequent changes to the representation will be easier.

Known Uses

Psion Software's EPOC32 'Word Editor' has two internal representations of a document. When the document is small, the editor keeps formatting information for the entire document; when the document is larger than a certain arbitrary size, the editor switches to storing information for only the part of the document currently on display. IntelliCorp's KEE package had two representations for objects on Lisp machines. Rolfe&Nolan's Lighthouse system has a 'Deal' class: by default this contains only basic data required for simple calculations; on demand, an instance extends itself reading the entire deal information from its database. The LOOM Virtual Memory system for Smalltalk [Kaehler 83] used multiple representations — one for objects completely in memory, and a second for leaf objects on secondary storage.

See Also

Basic Type with Object Wrapper can provide dynamic multiple representations for simple objects , but exposes the different representations to the client programmer. The **Bridge** pattern describes how objects can change their representations [Gamma+94]. Some representations can use explicit **Compression**, or can be stored in **Secondary Storage** or **Read-only Storage**. Changing to lower-overhead representations is an example of **Doing the Decent Thing.**

Variant: Basic Type with Object Wrapper Pattern

How can you allocate space for lots of small objects?

You have lots of small objects, each containing very little information—perhaps as little as one bit—but the minimum *memory requirement* of a class is quite large—perhaps several bytes. For example the Strap-It-On™'s artificial intelligence application (StrapThink) program has enormous numbers of entities, each representing a single truth value, and the Strap-It-On™ doesn't have enough memory to store each as a single object.

Therefore: *Use a compressed representation for storage and wrap it in an object for computation.*

Use basic data types (integers, characters, bits) and simple structures (arrays, records) to represent multitudinous tiny data items. When you need to process this data, wrap each item you need to process in a first-class object, and use the object to process the data. When you've completed processing, discard the object and recover and store the basic data once again.

Consequences

This *reduces memory use* enormously, especially in languages where basic types can be stored by value in other structures but where objects can only be stored by reference. **However:** wrapping basic types into objects requires additional code, *increasing programmer effort,* and the conversions between wrapped and unwrapped types must be *tested*. The unwrapped objects don't support object identity in the same way as the wrapped ones, and the two representations are inconsistent*, globally increasing the complexity* of every client which needs to use the data, *reducing design quality,* and requiring *more programmer discipline* to use each representation correctly.

Known Uses

For numbers, booleans and characters, Java supports both basic types (int, float), and object wrappers (Integer, Float). Programmers typically use the basic types for storage and the object versions for complicated operations. Unfortunately, Java collections store objects, not basic types, so every basic type must be wrapped before it is stored into a collection. Psion's EPOC32 implements strings using byte arrays; however operations on individual characters have a separate Char class supporting many more operations than the C++ primitive char. Garnet's virtual aggregates allow a single widget to display large amounts of bulk data by wrapping the data in the widget [Myers90].

See Also

Dynamic Multiple Representations can provide a common interface to different implementations of an object, supporting a common interface but requiring more overhead per object. **A Basic Type With Object Wrapper** is a special-purpose **Adapter** that provides a more complex interface for a basic type. The **Visitor** and **Presenter** [Vlissides 96] patterns can provide behaviour for collections of basic types without having to make each basic data item into an object.

Major Technique: Compression

How can you fit a quart of program and data into a pint pot of memory?

You're programming for a small system. Standard representations of code and data *require too much memory*, and there's no straightforward way to change structures involved to be smaller nor to discard any of the data you have. For example, the Strap-In-On wrist-mounted PC needs to store a large amount of executable code for its operating system and applications.

Therefore: *Use compression to reduce the memory required to store the information.*

Store the information in a compressed form and uncompress it when you need to access it. There are a wide variety of compression algorithms and approaches you can choose from, each with different space time trade-offs. In particular there are ad-hoc techniques such as run length encoding and byte codes, and larger scale algorithmic approaches based on finite state automata.

Consequences

Your *memory requirements* decrease as a result. **However**: the compression scheme will make the program more complex, requiring *programmer effort* to implement and *discipline* to use. The decompression process *reduces time performance* and may *require extra temporary memory*, increasing the possibilities for failure. The compression process also *takes time*, although this time can often be spent when the program is written rather than at run-time.

Known Uses

Compression is used widely through the industry. Operating systems use compression to store more information on secondary storage, communications protocols use compression to transmit information more quickly, and programming languages use compression to reduce the memory requirements of programs.

See Also

Sharing can reduce the memory requirements when the same information is replicated. **Byte Coding** and **String Compression** can compress programs or strings while still providing random access to the compressed information. **File Compression** can greatly reduce the memory required by large data items. **Short Names** can be used to compress program source code.

Sharing Pattern

Also Known As: indirection, normalisation.

How can you compress repeated information?

Sometimes the same information occurs many times throughout a program, increasing the program's *memory requirements.* The same procedure or library may be copied for use in different places in a program, or the same data may be reproduced wherever it is needed. These duplicates all use up scarce memory space. For example, the Strap-It-On user interface design includes many icons of the company's bespectacled founder.

Therefore: *Store information once, and share it everywhere it is needed.*

Analyse your program to determine which information is duplicated, and which can be safely shared. Any kind of information can be duplicated — images, sounds, multimedia resources, fonts, character tables, as well as application data. Code can be also duplicated, for example different subsystems may have their own copies of library classes or functions.

Once you have found common information, check that it can be shared. In particular, check that it never needs to be changed, or that all its clients can cope whenever it is changed. Then modify the information's clients so that they all refer to a single shared copy of the information, typically by accessing the information at another level of indirection. For example, the Strap-On-On PC really only needs one copy of Our Founder's bitmap. This bitmap is never modified, so one copy of the bitmap can be shared everywhere it is needed in the system. If the shared information can be discarded by its clients, you may need to use reference counting or garbage collection so that it is only released once it is no longer needed anywhere in the program.

Consequences

Judicious sharing can reduce a program's *memory requirements*, while increasing its locality. **However:** *programmer effort* and *discipline* is required to design programs to take advantage of sharing. Sharing can introduce many kinds of *aliasing problems*, especially when read-only data is changed accidentally [Noble 1998], and so can increase *testing costs*.

Known Uses

Lisp, Smalltalk, and Self all use an internal hash table so that read-only symbols are stored only once, and shared wherever they are used. The Unix utility **xstr** extracts string literals from C source code, removes duplicates, and arranges the strings so as to share substrings where possible. Modern C++ compilers share template instantiations to avoid code bloat.

See Also

Shared things are often **Read-only**, and so often end up as segments on **Secondary Storage**, getting updated by **Copy-on-Write**. The **Flyweight** pattern [Gamma+94]describes how to make objects read-only so that they can be shared safely. Objects may need to be moved into a **Single Place** for modelling reasons, too [Noble 1997].

Byte Coding Pattern

How can you compress machine instructions?

You have program code that must be present for the program to execute, increasing the program's *memory requirements*. How do you reduce its storage requirements while still allowing the program to execute? Simply using **File Compression** on the whole executable would require the whole executable to be uncompressed in one piece and the uncompressed version to be stored temporarily in main memory or on **Secondary Storage**; this is often unfeasible. For example, the Strap-It-On PC has a large amount of executable code and very little available temporary storage; it cannot execute programs from secondary storage.

Therefore: *Use byte codes to store the instructions, rather than full machine language.*

Rather than use machine-level instructions, compile the source code into an intermediate form, designed to occupy a minimum space. Execute the program using an interpreter (a 'virtual machine') which interprets these byte codes. For example the Strap-It-On compiles many programs to byte codes, and includes an interpreter which executes the byte codes from main memory.

Consequences:

Byte codes occupy much less space than native machine codes, reducing the program's *memory requirements*. A single byte code set may be *portable* across different machine architectures. **However:** byte codes are typically slower than native machine instructions, reducing *time performance* because they must be decoded and interpreted. In some cases a well-designed byte code set may provide better overall performance, when the time to load the codes over a slow network is included. Byte codes are quite easy to understand and encode, so they while they require some *programmer effort* to implement and *discipline* to use, it is not a great cost. You have to *test* the byte code compiler and interpreter, but these tests are quite straightforward.

Known Uses

Many compiler implementations compile to byte codes: most Smalltalk variants, Squeak!, BCPL for the BBC Micro, etc. Java and Inferno always compile to a byte code format; minimising code size and transmission time, and incidentally ensuring these systems' cross-platform portability. The SubArctic toolkit uses byte codes to store the constraints used for interface layout [Hudson+96].

See Also

Byte codes can be interpreted by a variant of the **Interpreter** pattern. **String Compression** uses similar techniques to compress strings. **File Compression** uses more powerful algorithms to achieve better compression ratios for large data items which do not need to be accessed randomly. **Short Names** is a quick and nasty alternative for compressing interpreted source, and can also be used within byte codes. The details of byte coding and byte code interpreters have been described well elsewhere. [Goldberg 83, Richards 77].

String Compression Pattern

How do you reduce the memory taken up by static text strings in a program?

Strings displayed to the user, database keys, keys to X window system resources and database query strings all take a significant amount of memory, increasing the program's *memory requirements*. Typically you will have many strings, all different from each other. You need to provide random access to support the typical string operations, but you don't want to put to much *programmer effort* into the problem.

Therefore: *use a standard automated string-compression technique, and use compilation tools to compress the strings.*

There are many simple string compression techniques which provide a simpler compressed representation for strings but still support common string operations easily. 'Nibble' codes use a 4-bit, 5-bit, or 6-bit representation for the most common characters; other characters are represented as an escape character followed by the full 8-bit representation. UTF8 is a representation of Unicode 2-byte characters designed for data transmission; it also provides good compression for Unicode text that is predominantly in a single 8-bit character set.

Consequences

Typically you get a reasonable compression for the strings themselves (40% or so), reducing the program's *memory requirements*. String compression is quite easy to implement, so it does not take much *programmer effort*. Operations on these compressed strings can execute almost as fast as operations on native strings, preserving *time performance*. **However:** the total compression of the program data isn't high, so the program's *memory requirements* are not greatly reduced**.** You have to *test* the compressed string operations, but these tests are quite straightforward.

Known Uses

Nibble codes were widely used in versions of text games for small machines [Blank+95]. Philip Gage used a similar technique to compress an entire string table [Gage97]. Early versions of MacWrite used compression ubiquitously, both in main memory and in text files [Bell+90].

See Also

File Compression uses more powerful algorithms to achieve better compression ratios for large data items which do not need to be accessed randomly. **Byte Codes** use similar techniques to compress instructions.

File Compression Pattern

How can you compress a large amount of bulk information?

A large amount of your program's *memory requirements* are for storing bulk information, such as text, images, sound files, or other multimedia data, to which you do not need random access. For example, the Strap-It-On PC needs to store a ten-minute video of the founder of the company, which will be shown on the 2in screen.

Therefore: *use an adaptive compression algorithm*

There are a number of standard adaptive compression algorithms, with Lempel-Ziv and LWZ being the most popular. Use one of these algorithms, either on a whole batch of data at once, or incrementally processing the data stream. Implementations of many algorithms are available publicly. For example, Strap-It-On PC stores the video in the standard MPEG compression format.

Consequences

Modern adaptive compression algorithms provide great compression ratios (reducing your *memory requirements*), are widely used, and are incorporated into popular industry standards for bulk data. **However:** they can require quite a large amount of *processing time*, and *require temporary memory* (primary or secondary) to store the results and to hold intermediate structures. It is difficult to access data store in compressed files randomly. The performance of compression algorithms can vary depending on the type of data being compressed, so you have to select your algorithm carefully. If you cannot reuse an existing implementation., you will need *programmer effort* to code up one of these algorithms, because they are quite complex. Some of the most important algorithms are *patented*, although you may able to use non-patented alternatives.

Known Uses

Lempel-Ziv and variant compression algorithms are an industry standard, evidenced by the many PKZip and gzip file compression utilities. These algorithms have also been incorporated into many image file formats such as GIF. File compression is also used architecturally in many systems. For example, Java uses ZIP compression for class files, Linux kernels can be stored compressed and are uncompressed when the system boots, and Windows supports optional file compression for each disk volume.

See Also

Text Compression [Bell+90] provides a good overview of most of the common algorithms, including algorithms used for non-textual data.

Short Names Pattern

How can you reduce the size of program source code, while ensuring it can still be parsed?

Sometimes you need to compress source code or scripting language programs, to reduce their *memory requirements*, but you want to keep the programs editable, and you don't want to expend much *programmer effort* in implementing compression. For example, the Strap-O-Matic supports StrapBasic, but it needs to keep the program source codes small.

Therefore: *use shorter names or sharing in the program source code.*

Shorter variable names occupy less space so you can shorten the names to shorten the code. In many languages literals occupy more space in source or byte codes than variables, so you can assign often used literals to variables and then replace the literals by the variables, effectively **Sharing** one literal in many places. You may be able to use a post-processor to perform these transformations automatically. More brutally, some languages simply restrict the length of identifiers permitted in the source code. For example, StrapBasic programs are restricted to using only twenty-six variables named "a" to "z".

Consequences

Short names are easy to use manually, requiring no *programmer effort,* and impose no *time* overhead. In languages where parsing is expensive, or names are rescanned repeatedly, programs using short names can load and execute faster than programs with longer names. Implementing a pre-processor to replace names with shorter or smaller versions is not difficult, as it is a source-to-source transformation, requiring little *programmer effort*. **However:** making these kinds of changes to a program can introduce errors, so they must be *tested*, and the resulting compression ratio isn't that high, so *memory requirements* are not greatly reduced. A program which uses short names has low *design quality*, because it will be hard to understand. It is completely evil to do this to source code someone else will have to maintain, so using short names breaks the most fundamental *programmer discipline*.

Known Uses

Early BASIC's, including the ZX81 Basic, restricted all identifiers to single alphabetic characters. Identifier compression programs were commonly used for BBC Basic to save space, and have now resurfaced for Java. In ZX81 Basic, variables required one byte while floating point literals required eight bytes, so common floating point literals would be stored in variables and reused. LaTeX uses similar techniques — common literals such as one, two, and minus one are coded as the macros '\@ne', '\tw@', and '\m@on'. Unix commands (and, famously, the `creat` system call) lack vowels for similar reasons.

See Also

Byte Coding is a more serious form of code compression.

This is the complete antithesis of **Simply Understood Code** [Gabriel].

Major Technique: Secondary Storage

What can you do when you have run out of primary storage?

Sometimes, your system's primary memory is just not big enough to fulfil your program's *memory requirements*. For example, many small systems need to provide word-processors. The Word-O-Matic™ word-processor for the Strap-It-On™ needs to be able to let users edit large amounts of text. It also supports formatting text for display or printing, not to mention spelling, grammar checking, table and chart formatting, and mail merge. Many small systems will not have enough main memory to store the output of a large mail merge, even ignoring the fact that the word processing software also needs to occupy some of that memory.

For most programs, there is usually some hope. Even in small systems, a program's *working set size* (the amount of memory a program requires to make progress) is usually smaller than the memory available, and is usually also a small fraction of the whole program. Many small systems also have some reasonably fast *secondary storage* — storage like disc, PCMCIA Cards, tapes, or networks — which is quite close to the primary storage but which the CPU cannot access directly. For example, although the Strap-It-On PC has only 1M of main memory, it comes with a CyberStrap™ which includes a 32Mb bubble memory store (along with interfaces for wrist-mounted floppy drives, etc.).

Therefore: *Use secondary storage as extra memory at runtime.*

Divide up your program and data into pieces. Load into main memory only those pieces of code and data that you actually need, and somehow replace them with more relevant pieces when they are no longer required.

The following variant patterns describe a number of different ways of dividing up the program, and loading and unloading the resulting pieces. The patterns have much in common, but differ in the following ways:

- What is divided up: the program code, the data, configuration information or some combination?
- Who does the division: the programmer, the system or the user?
- Who does loading and unloading: the programmer, the system or the user?

Consequences

Being able to use *secondary storage* can be like getting a lot of extra memory for free — it greatly reduces your program's *primary memory requirements.* **However:** the secondary storage must be managed, and information transferred between primary and secondary storage. This management has a *time performance* cost, and may also cost programmer *effort* and *programmer discipline*, impose *local restrictions* to support *global mechanisms,* require hardware *or operating system support,* and reduce the program's *usability.*

See Also

The following patterns form a sequence starting with simple patterns which can be implemented *locally* with *programmer discipline*, and finishing with complex patterns which require *global hardware or operating system support* but *reduce the programmer discipline* required. The simplest pattern is **Program Chaining**, followed by **Data Chaining**, **Autoloading**, **Resource Files**, **Segmentation**, and the most complex pattern, **Paging**.

Read-only pieces of program or data can be deleted from memory without having to be saved back to secondary storage.

Program Chaining Pattern

Also known as: exec, multiple passes, multiple phases

What's the easiest way to use secondary storage?

Some programs are big, but can be broken into independent pieces, where only one piece is used at a time, where the control flow between pieces is quite simple, and where the *memory requirements* for each piece are less than the *memory requirements* for the program as whole. For example, Word-O-Matic divides conceptually into three independent pieces: a text editor that allows the user to enter text, a spelling and grammar checker to check the text, and a text formatter for final printing.

Therefore: *split the program up into independent pieces, and chain each piece.*

Each piece of the program can be turned into an almost independent program, called a *phase*, which runs individually and then invokes (or 'chains') the next *phase* when it finishes executing. For program chaining, the programmer is responsible for dividing up the program into phases; then either the programmer (in designing the program) or the user (at runtime) is responsible for loading and unloading each phase. The data used by each phase needs to live somewhere: in main memory, in secondary storage that is reloaded into each phase, or in pipes between phases. For example the Word-O-Matic program could be divided into three phases, editing, checking and formatting, each invoked by the user.

Consequences

Program chaining uses *secondary storage* to reduce a program's *memory requirements* and, being simple to implement and easy to use, costs little *programmer effort*. **However:** it is really important that the various pieces of the program are *independent*, and ideally that the division makes some kind of sense to the user. Dividing a large program into a good set of phases can be quite difficult, so a multi-phase program can require significant *programmer effort* to design, and *local complexity* to call the appropriate phase. If you have too many phases, the cost of loading each phase and transferring data can dominate the program's *run time performance*; this is also a problem if the control flow between different phases is complex — if the program swaps quickly between two different phases. Making the user manage the phases reduces the program's *usability*.

Known Uses

BBC BASIC had a 'CHAIN' command especially for invoking different phases; UNIX's 'exec' command can be used similarly. Many microcomputer adventure games used program chaining. Two good examples are Apple II's Wizardry which had an edit phase and a play phase, and the original BBC Elite which had a flight phase and a trading phase. Unix compilers have traditionally been implemented as a number of phases linked by pipes: Algol-68 compilers routinely had four to six phases [HOPL-II].

See Also

Autoloading, **Segmentation**, and **Paging** are more complex alternatives to this pattern. Each phase can be stored on **Secondary Storage**, using **Compression**. **Data Chaining** describes a similar technique for handling a program's data.

Data Chaining Pattern

How can you process a large amount of data in secondary storage?

Sometimes programs themselves are quite small, but need to process a large amount of data —the *memory requirements* mean that the program will fit into main memory, but the data will only fit into secondary storage. For example the input and output files of Word-O-Matic's text Formatter may exceed the capacity of the main memory when formatting a large book. Dividing the program up into smaller phases (as in Program Chaining) can reduce the memory required by the program itself, but this doesn't help reduce the memory requirements for the input and output.

Therefore: *Process one small portion of the data at a time, reading the inputs and writing the results to secondary storage.*

The input and output data files can be divided into a number of sub-files, and each sub-file can be processed individually. Once each sub-file has been processed, all the output sub-files can be combined into a final result file, presumably processing the files from secondary storage and perhaps using a separate program phase. For example Word-O-Matic's text formatter could process the book one chapter at a time. Generally the user must subdivide the data into sub-files and arrange to load each sub-file appropriately.

Consequences

In general, data chaining is the complement of program chaining, and its forces, benefits, and liabilities are similar to that pattern — using *secondary storage* to reducing program's *memory requirements* at a reasonable cost in *programmer effort.* **However:** *Programmer effort* is required to design the program so that each sub-file must be able to be processed *independently* — in particular, sub-files processed early shouldn't depend upon sub-files processed later in the run. Often extra information (about other sub-files) needs to be added to each sub-file so that they can be processed independently. Data chaining will provide slower run-*time performance* than processing all the input in one piece. Making the use deal with the sub files reduces the program's *usability*.

Known Uses

Many traditional programming languages are designed to be compiled in this way. For example C and C++ languages can be compiled one file at a time, and the sub-files (object files) combined with a single linking phase after all the compilation phases. C and C++ also force you to include header files appropriate to the piece of code before the related code. This pattern is ubiquitous in old-time batch tape processing.

See Also

Resource Files, and **Paging** are more complex alternatives to this pattern. Each sub file can be stored on **Secondary Storage**, using **Compression**. **Data Chaining** describes a similar technique for handling a program's code.

Autoloading Pattern

How can you manage a large program with lots of optional pieces?

Some big programs are really small programs much of the time — the program's *memory requirements* are much greater than its working set size. For example Word-O-Matic's text formatter only needs a very few formatting rules: how to format paragraphs left-justified, how to format headings, and how to format pages. Of course to be general the text formatter needs a lot more facilities: tables, running headers and footers, double sided or two column pages, book, article, and thesis styles, and so on. But as in many programs the extra facilities are almost never used, increasing the program's *memory requirements* even when unneeded.

Therefore: *Load each piece from secondary storage as it is needed.*

The programmer must divide the program into a *main program* and a collection of independently loaded *packages*. The *packages* are associated with *stub routines* in the program. The main program is loaded and starts running. When it needs to use a facility in a package, the main program calls the stub routine, which loads the appropriate package, and then redefines itself to call the package directly. For example, the core of Word-O-Matic's text formatter is the main program, and the special formatting commands are stub routines. When the formatter encounters a special formatting command, it should call a stub to load the appropriate formatting rules automatically.

Consequences:

The program will *require less memory* because some of its code is stored on *secondary storage* until needed. The program will start up quicker, increasing *time performance* as only the small main program needs to be loaded initially, and can begin running with *less memory* than would otherwise be required. Because each package is small, subsequent packages can be loaded in quickly, without pauses for changing phases (as in the program chaining pattern). Because packages are loaded automatically when their stub routine is called, autoloading is transparent, so no *programmer discipline* is require to take advantage of it. Also the user can also be quite unaware that the program is divided and autoloading various packages, which increases the program's *usability* compared to manual program chaining. **However:** *Programmer effort* is needed to divide up the program into packages. Packages are necessarily never unloaded, so the program's *memory requirements* will steadily increase, and it can still run out of memory unless any given run uses only a small part of the whole thing. Autoloading doesn't help much when the program consists of a number of independent phases. *Programmer effort* is also needed to implement the autoloading mechanism, which can often be reused across programs, or it may be provided by the *operating system.*

Known Uses

Many Lisp systems including GNU Emacs use autoloading, as does the Tcl interpreter. Many modern operating systems (Solaris, MS Windows, etc.) support Dynamically Linked Libraries, which are only loaded when any part of their code is first executed.

See Also

Program Chaining is a simpler alternative to this pattern, which is applicable when the program can be divided into independent phases. **Resource Files** and **Paging** are more complex alternatives to this pattern. Each package can be stored on **Secondary Storage**, using **Compression**.

Virtual Proxies [Gamma+94] can be used to autoload individual objects.

Resource Files Pattern

How can you manage lots of static data?

Sometimes a program's *memory requirements* include space for a lot of static data, but the program only uses a small amount at any one time. For example Word-O-Matic needs static data such as window layouts, icon designs, font metrics and spelling dictionaries. Much of this information may be requested at any time, but when requested it is often needed only for a short time. If the information is stored in main memory it will increase the program's overall *memory requirements*. Although the data is *read-only*, often it cannot be stored directly in **Read-only** main memory because the main memory does not have enough capacity. The data can't simply be divided up into phases, sub-files or packages, because several packages may be needed at once and simply autoloading everything would fill up the memory.

Therefore: *Store the data in resource files, which can be loaded and unloaded as necessary.*

Typically the static data is divided up into files by the programmer. When a piece of data is required, the program calls a special routine. This routine loads the data if it not already in memory then returns it to the program. Once finished with the data, the program calls a routine that releases the data, after which it may be automatically unloaded to make room for more data. For example, all of the word-processor's window layouts, icon designs and text strings can be stored in resource files. When they are required the word-processor can retrieve the data from the resource file, yet the memory can be reused when the data is no longer required.

Consequences

The static data doesn't clutter primary storage, reducing the program's *memory requirements*. This also makes it easy to change the data without changing the program (e.g. to support multiple language strings), increasing the program's *design quality*. **However:** this requires *programmer discipline* to place resources into the resource files, and to load and release the resources correctly. Loading and unloading resource files reduces the program's *time performance* somewhat. Resource files also need more *programmer effort* to implement than simple **Autoloading**, because you need some mechanism to unload (and reload) the resources. It's best if the *operating system* environment provides this support.

Known Uses

Virtually all Apple Macintosh and MS Windows GUI programs use resource files to store GUI resources. VAX/VMS uses resource files to store program messages.

See Also

Data Chaining and **Autoloading** are simpler alternatives to this pattern, while **Paging** is a more complex alternative. Each resource file can be stored in **Secondary Storage** or **Read-only Storage** using **Compression**. **Segmentation** describes a similar technique for handling code.

Segmentation Pattern

How can you manage lots of code?

Sometimes a program's *memory requirements* include space for a lot of code, but the program only uses a small amount of code at any one time, irrespective of whatever phase it might be using. For example Word-O-Matic will need to display windows, open files and so on, whether the user is editing text, checking spelling or formatting output. Because this code may be required at any time it can't simply be organised into phases which are executed sequentially. However keeping it all in main memory (even in **Read-only Storage,** since code is typically *read-only)* would exceed the program's *memory requirements*.

Therefore: *Divide up the code into separate segments, which can be automatically loaded and unloaded as necessary.*

Typically the program is divided up into segments by the programmer. When a segment is required, the program calls a special routine that loads the segment if not already in memory, allowing the programmer to call routines in the segment. When the program is finished with the code, it releases the segment so that the memory can be reused. Although this division is often manual, some systems provide automatic support both for the segmentation process and loading and unloading the segments. For example, all of Word-O-Matic's utility code could be divided up into segments that are then loaded when they are required.

Consequences

Code segmentation is complementary to storing data in resource files, resolves similar forces and offers similar benefits and liabilities. In particular, unneeded code doesn't clutter primary storage, reducing *memory requirements.* **However:** this needs *programmer effort* to work out a logical segmentation of the program, then *programmer discipline* to place code into segments, and to load and release the segments correctly. Segmentation reduces *time performance* compared with **Program Chaining** or **Autoloading** because every call needs to be indirect. Segmentation takes more *programmer effort* to implement than simple **Program Chaining** or **Autoloading**, because you need some mechanism to unload (and reload) the resources. Again, it's best if the *operating system* environment provides support.

Known Uses

Some PDP-11 compilers could segment programs automatically, so could BBC BCPL compilers (segmenting across ROMS) and the Macintosh.

See Also

Program Chaining and **Autoloading** are simpler alternatives to this pattern, while **Paging** is a more complex alternative. Each resource file can be stored in **Secondary Storage** or **Read-only Storage** using **Compression**. **Resource Files** are a similar technique for handling data.

Paging Pattern

How can you provide the illusion of infinite memory?[1]

Some programs' *memory requirements* are too big to fit into memory, but there just aren't any obvious ways to divide up the program. They program may exhibit strong *dynamic* locality, but this may bear little relation to the *static* structure of the code. This is especially the case when OO programming is used, what with inheritance and everything. For example, Word-O-Matic must provide a comprehensive help system, especially regarding its two-button chord-input keyboard; however the help information is accessed randomly and is too large for RAM.

Therefore: *Use demand paging to transparently run the program directly from secondary storage.*

Pick a page size, and arbitrarily (and mechanically) divide up the program and available memory into page-sized chunks. Start the program by loading page one into memory, and start running the program, somehow catching every memory access. When the program needs to access something that isn't in main store, load that into an empty page frame (memory page), fix up the address and carry on. If there aren't any available page frames then empty one, writing its content back to secondary storage store if it has been modified. Make an interpreter or data manager, or build hardware, to hide the paging from the user program. For example the hypertext help system for the two-button chord-input Mouse-O-Keyboard on the Word-O-Matic uses paging to provide access to its help data.

Consequences

Paging is the ultimate escape of the memory-challenged programmer. The programmer is **much less** aware of paging than any other technique, since paging provides the illusion of essentially infinite memory — the program's *memory requirements* are no longer a problem. Paging tends to increase other aspects of a program's *design quality*, because memory requirements are no longer an overriding issue. Paging needs little *programmer effort* or *programmer discipline*, because it doesn't need a logical decomposition of the program, and handles phases quite well – the old phase is paged out as the new phase is paged in. Paging can make better use of the available memory where the working set is distributed over the logical modules of the program. **However:** paging reduces a program's *time performance,* because the routine management is slower than segmentation or other manual memory management techniques, and usually needs fast secondary storage to perform well. Of course 'fast' is a relative term; lots of systems have used floppy disks for paging. Paging encourages major code bloat, so a program's *memory requirements* will increase in paged systems, expanding to fill the secondary storage available. Paging requires no *local* support from within programs, but requires low-level *global* support, often provided by the *hardware and operating system*, or an interpreter or data manager.

Known Uses

Infocom games implemented a paged interpreter on machines like Apple-IIs and early PCs [Blank+95]. Lots of operating systems provide virtual memory, including UNIX and MS Windows.

See Also

The other patterns in this section — **Program Chaining, Data Chaining, Autoloading, Resource Files,** and **Segmentation** — provide alternatives to this pattern. Generally these alternatives will have better *time performance* and less *memory requirements*, and have a more *local* impact, and require less *hardware and operating system support* than paging, but will require more *programmer effort* and *programmer discipline* to use. Paging can also use **Copy on Write** to optimise access to *read-only* storage, and can be extended to support **Sharing.**

---

[1] *Sometimes the program is just too big, to complex, or you are too lazy to segment, subdivide, chain, phase, slice, dice, vitamise, or food process the code any more. Why* **should** *programmers have to worry about memory! Infinite memory for all is a* **right**, *not a privilege! Those memory protection society guys are just* **no-life-losers**!

Major Technique: Read-only Storage

What can you do with read-only code and data?

Programs often have lots of *read-only* code and data. For example, the Word-O-Matic™ word-processor needs a large amount of executable code and large master dictionary files for its spelling checker, which it never changes. Storing all this information directly will use up precious main memory, displacing memory capacity from data that actually needs to change and increasing the *memory requirements* of the program as a whole.

Therefore: S*tore read-only code and data in read-only storage.*

Divide your program and data into those portions which can change and those which never change. Store the immutable portions in read-only memory or secondary storage, and arrange to re-associate them with the changeable portions at run-time. The most common situation is where read-only code has related writable data. For example Word-O-Matic's code and master dictionary is stored in read-only secondary storage, while user documents are stored in writeable storage.

Consequences

This pattern trades off writable storage for *read-only* storage, reducing the program's *memory requirements* for main storage. Read-only storage is cheaper than writable storage, in terms of financial cost, power consumption and reliability. Although read-only code and data must be copied from *read-only secondary storage* to main memory, they can be deleted from memory without having to be saved back to secondary storage. Sharing read-only data between programs further reduces the *memory requirements* of the system as a whole. **However:** *programmer effort* is needed to divide up the program, and them *programmer discipline* to stick to the division.

Known Uses

Read-only storage is ubiquitous. Many linkers support the separation between read-only and read-write storage. One of the more famous (or infamous) examples is the split Instruction (read-only) and Data (read-write) address spaces in the PDP-ll. Embedded systems use (cheap) PROMs or EPROMs for read-only code, saving scarce expensive RAM for dynamic data. The Unix **xstr** utility moves string literals out of the writable data segment into external files that can be stored in read-only secondary storage.

See Also.

Data in read-only storage can be changed using **Copy-on-write** and **Hooks. Copy-on-write** and **Hooks** also allow some kinds of infrequently changing (but not constant) data to be moved to read only storage. Because the contents of read-only storage cannot be changed, it is a good candidate for **Pre-Initialisation**. Data in read-only storage is a good candidate for **Sharing** between various programs, or for moving to **Secondary Storage**.

Copy-on-Write Pattern

How can you change stuff in read-only storage?

Sometimes you need to change portions of the program which are in *read-only* storage. For example, Word-O-Matic's document templates are read-only, but sometimes users want to change them.

Therefore*: Copy read-only data when you need to change it, and use the copy in future.*

When an attempt is made to change a portion of read-only storage, the system should copy that portion into writable storage and then make the change to the copy. For example, although Word-O-Matic's template in ROM files cannot be changed, if a user tries to change them they are automatically copied and moved into RAM. The amount of data which must be copied will depend upon the granularity of the data structures in the program's design.

Consequences

Copy-on-write gives programmers the illusion that data stored in *read-only* storage can be changed. As a result, infrequently changed data can be moved into *read-only* storage, reducing the program's *memory requirements.* Once Copy-on-write has been implemented it requires little *programmer discipline* to use, as programmers don't need to be directly aware of it. **However:** Copy-on-write requires *programmer effort* or *hardware or operating system* support to implement, because the system needs to intercept writes to read-only storage (typically by detecting exceptions), make the copy and then continue the write as if nothing had happened. Because Copy-on-write is easy to use, it decreases *local complexity* at the cost of providing a *global mechanism*. Intercepting attempted writes can slow down access to read-only storage, and is certainly slower than writing directly to writable storage, so this may reduce *time performance*. Copy-on-write can cause problems for object identity if the identity of the copy and the original storage is supposed to be the same. Copy-on-write can also lead to lots of copies of the same thing cluttering up the system, ultimately increasing the system's *memory requirements.*

Known Uses

Unix supports Copy-on-write in its paging system. RogueWave's Tools.h++ library uses it for its CString class. Microsoft Word sometimes uses it for managing shared files in network installations.

See Also

Hooks provide a complementary technique for changing the contents of read-only storage**.** Always accessing read-only storage via a **Proxy** can hide the copy from the programmer [Gamma+94]. **Paging** systems often use Copy-on-Write**.** Copy-on-Write is a **Multiple Representation**, and uses **Sharing** when the data isn't copied.

Hooks Pattern

How can you transparently replace or extend the contents of read-only storage

Sometimes it's not just enough to change *read-only* storage; you really need to replace or extend its contents in such a way that other users of the storage will see your changed version instead. For example, all versions of the Word-O-Matic code stored in the Strap-It-On's ROM are very buggy, so bug fixes need to be incorporated into the running system. It's not practical to copy the Word-O-Matic ROMs in their entirety into RAM, because the *memory requirements* mean that this would leave no space for anything to edit.

Therefore: *Link read-only code and data together through hooks in writable storage.*

When designing your program or data structure, arrange that the *hot spots* [Pree 94] that link important portions of your program together are stored as hooks in writable main storage. Always access parts of your program indirectly through the hooks. To update a portion of your program you can copy and modify just that portion, store the modified copy in writable store, then set the hooks to point to the modified portion. The modified portion should be able to call other portions of the program, again by indirection through the hooks.

Consequences

Hooks let you extend *read-only* storage, and by making *read-only* storage easier to use, can reduce the program's writable *memory requirements*. Providing good hooks increases the *quality* of the program's design, making it easier to extend in future. **However:** hooks require *programmer discipline* to design into programs and then to ensure they are used. They also increases the *testing* cost of the program, because the hooks have to be tested to see if they are called at the right times. Indirect access via hooks is slower than direct access, reducing *time performance*; and the hook vectors take up valuable writable storage, slightly increasing *memory requirements*. Hook vectors are great places to attack system integrity, as any virus writer will tell you.

Known Uses

The Mac, BBC Micro, and IBM PC ROM BIOS's are all reached through hook vectors in RAM. Emacs makes great use of hooks to extend its read-only code. NewtonScript allows objects to inherit from read-only objects, using both hooks and copy-on-write. Processor interrupt vectors are hooks for handling processor exceptions and interrupts.

See Also

Copy-on-write is a complementary technique for changing information in **Read-only** storage, and **Copy-on-write** and Hooks can often be used together. Using hooks in conjunction with **Read-only** storage is a special instance of the general use of hooks to extend systems one cannot change directly. See, for example, the **Observer** pattern [Gamma+94].

Pre-initialise Data Pattern

How should you initialise data in primary storage?

Sometimes programs need to initialise main memory structures by copying from *read-only* storage. For example, Word-O-Matic needs to load all sorts of miscellany including spelling exception dictionaries, hyphenation tables, and cute pictures of paper-clips to provide an extra interface to a perfectly usable help system. One simple solution is to write code that reads through a data file and initialises each datum individually based on the file's contents. While this solution does the job, it requires code to perform the initialisation, which takes *programmer effort* to write, and increases the program's *memory requirements*.

Therefore: *Pre-initialise data from read-only store, and load it directly.*

If possible, load the data directly from *read-only* memory into main store, in the format that you actually need it. Avoid any intermediate processing of the data.

Consequences

Loading data directly from *read-only* storage eliminates the file-handling code which would otherwise be required to initialise the data structures, doing away with the *programmer effort* needed to write that code, the *memory requirement* to store the code, and increasing the program's initialisation *time performance* and *design quality*. **However:** in some cases the initialised data may take more space than code to initialise the structures programmatically, in which case loading pre-initialised data would increase the program's *memory requirements*.

Known Uses

Yacc and Lex generate and initialise parse tables as C constants rather than loading them from a file. Typical compilers do constant folding at compile time, so the C "(1 << 8)" is actually stored as an integer. LaTeX, Emacs, Smalltalk, and APL allow loaded and initialised systems to be dumped directly to disk so that all initialised data can be loaded along with the executable file.

Acknowledgements

The authors wish to thank John Vlissides, our conference shepherd, for his valuable and exacting feedback on earlier drafts of this work.

References

[Aho+82] Aho, Hopcroft and Ullman (1982) *Data structures and algorithms* Addison-Wesley ISBN 0-201-00023-7

[Alex 77] Alexander C. (1977) *A Pattern Language: Towns/Buildings/Construction*. Oxford University Press.

[Adams95] Adams S. (1995) *Functionality a-la-carte*. In *Pattern Languages of Program Design.* Coplien and Schmidt, eds, Addison-Wesley.

[Auer+95] Vlissides, Coplien and Kerth (1995), *Pattern Languages of Program Design 2,* Addison Wesley ISBN 0-201-89527-7, *Chapter 2: Lazy Optimisation: Patterns for Efficient Smalltalk Programming.*

[Bell+90] Bell, Cleary and Witten (1990) *Text Compression*. Prentice-Hall.

[Blank+95] Blank and Galley (1995) *How to Fit a Large Program Into a Small Machine*, available as http://www.csd.uwo.ca/Infocom/Articles/small.html

[Brooks75] Brooks, F (1975) *The Mythical Man Month*, Addison-Wesley ISBN 0-201-00650-2

[Gamma+94] Gamma, Helm, Johnson and Vlissides (1994) *Design Patterns*, Addison-Wesley ISBN 0-201-63361-2

[Gilb88] Gilb T (1988) *Principles of Software Engineering Management*, Addison-Wesley ISBN 0-201-19246-2

[Gabriel96] Gabriel, R.P. (1996), *Simply Understood Code.* Availible as http://www.c2.com/cgi/wiki?SimplyUnderstoodCode

[Gage97] Gage, P (1997) *Random access data compression* C Users Journal Volume 15 number 9, September 1997.

[Goldberg+83] Goldberg, A, and Robson, D. (1983) *Smalltalk: The language and its Implementation*. Addison-Wesley.

[Hudson+96] Hudson, S.E., and Smith, I. (1996) *Ultra-Lightweight Constraints*. In UIST Proceedings.

[Kaehler+83] Kaehler T., and Krasner, G. (1983) *LOOM — Large Object-Oriented Memory for Smalltalk-80 Systems.* In *Smalltalk-80: Bits of History, Words of Advice*. Glen Krasner, ed. Addison-Wesley.

[Lindsey 93] Lindsey C.H. (1993) A History of ALGOL 68. In *History of Programming Languages-II.* ACM Press.

[MacNiel95] MacNeil, J. and Proudfoot, D. (1995) *The Rainy Day Fund.* Technical Note CPX_TN #30, Pictorius Inc.

[Myers+90] Myers, B.A. *et. al. (*1990) *Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces.* IEEE Computer, 23(11).

[Nilson95] Nilson, K (1996) *Issues in the Design and Implementation of Real-Time Java*, available as http://www.sys-con.com/java/iss1/real.htm

[Noble97] Noble, J. (1997) *Patterns for finding objects within designs.* In TOOLS Pacific 25.

[Noble98] Noble, J., Vitek, J., and Potter, J. (1998) *Flexible Alias Protection.* In ECOOP Proceedings.

[Pree94] Pree, W.(1994) *Design Patterns for Object-Oriented Software Development.* Addison-Wesley.

[Richards+77] Richards, M. and Whitby-Stevens. R (1977) *BCPL: The language and its compiler.*

[Vlissides96] Vlissides, J (1996) *The trouble with Observer*. C++ Report.