Small User Interfaces

These patterns are part of an ongoing project to capture and document techniques for the design and construction of systems that must function under tight memory constraints. Some patterns from this project are already published in the book *Small Memory Software* in the Addison-Wesley Software Patterns Series.

This paper contains the following patterns:

- Make the User Worry
- Fixed User Memory
- Variable User Memory

The full chapter will include three additional patterns, together with more structure and a bibliography; the introductory 'Major Technique' pattern includes an overview of all the patterns involved.

Throughout this paper we refer to pattern names in **BOLD CAPITALS**, and to 'forces', the major factors to consider in choosing patterns, in *Italics*. Unless otherwise qualified, the pattern names refer to those in this chapter or in the book *Small Memory Software*.

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

You can find further information about the Small Memory Software project on the web at: http://www.smallmemory.com

Credits

We'd like to thank all those who have reviewed, comment on, or shepherded these patterns. Particular thanks go to Linda Rising, our Europlop 2000 shepherd, for her usual incisive comments!

Major Technique: Make the User Worry

How can you manage memory in an unpredictable interactive system?

- Memory requirements can depend on the way users interact with the system.
- If you allocate memory conservatively, the systems functionality may be constrained.
- If you allocate memory aggressively, the system may run out of memory.
- The system needs to be able to support different users who will use the system in quite different ways.
- Users need to perform a number of different tasks, and each task has different memory requirements.
- The system may have to run efficiently on hardware with greatly varying physical memory resources.
- It's more important that the system has sufficient capacity than that the system is easy to use.

In many cases, especially in interactive systems, memory requirements cannot really be predicted in advance. For example, the memory requirements for the Strap-It-On PC's word-processing application Word-O-Matic will vary greatly, depending the features users choose to exercise — one user may want voice output, while another may choose a large font for file editing, and a third may require a large amount of clip art.

The memory demands of interactive systems are unpredictable because they depend critically on what users choose to do with the system. If you try to produce a generic memory budget, you will over-allocate the memory requirements for some parts of the program, and consequently have to under-allocate memory for others.

For many interactive systems, providing the necessary functionality is more important than making the functionality easy to learn or to use. Being able to use a system to do a variety of jobs without running out of memory is sufficiently important that you can risk making other aspects of the interface design more complicated if it makes this possible. This is especially important because a system's users presumably know how they will use the system when they are actually using it, whereas the system's designer may not know this ahead of time.

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

Design a conceptual model of the way the system will use memory. Ideally this model should be based on the user's conceptual model of the system, but should provide extra information about the system's memory use. Express the memory information in terms of the objects that users manipulate, rather than the objects used directly in the implementation of the system.

Expose this memory model in your program's user interface, and let users manage memory allocation directly – either in the way that they create and store user interface level objects, or more coarsely, balancing memory use between their objects and the program's internal memory requirements.

For example, Word-O-Matic makes the user take responsibility for the system's use of memory. On request, Word-O-Matic displays the amount of memory it has available, and allows users to choose how that memory should be allocated. If a user wants to store a larger amount of clip-art they well need to allocate sufficient memory to store the clip-art that they need. Word-O-Matic uses all the otherwise unallocated space to store the document, so a user subsequently needs to work on a long document, the user will need to reduce the amount of memory allocated to clip-art.

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 can be quarantined within the **MEMORY BUDGET**, so the memory use of the system as a whole is more predictable. Some user interfaces can even make the user worry about *memory fragmentation*.

However: Users now have to worry about memory whether they want to or not, so the system is less *usable*. Worrying about memory complicates the design of the system and its interface, making it more confusing to users, and distracting them from their primary task. Given a choice, users will choose systems where they do not have to worry about memory. You have to spend *programmer effort* designing a user interface that makes the memory model visible to the user.

❖ ❖ ❖

Implementation

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

- Constantly display the amount of free memory in the system.
- Provide tools that 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, or as the user allocates lots of data.
- Let the user choose what data to overwrite or delete when they need more memory.
- Show the memory usage of different components in the system.
- Tell users how their actions and choices affect the system's memory requirements.

Here are some further issues you should consider when making users worry about memory allocation:

1. Supporting Different Kinds of Users. Different users can differ widely in the roles they play with respect to a given system, and often their memory use (and interest in or capability to manage the system's memory use) depends upon the role they play. For example, the users of a web-based information kiosk system would have two main roles with respect to the system: a casual inquirer trying to obtain information from the kiosk, and a kiosk administrator configuring the system, choosing networking protocol addresses, font sizes, image resolutions and so on. The casual inquirer would have no interest in the system's model of memory use, and no background or

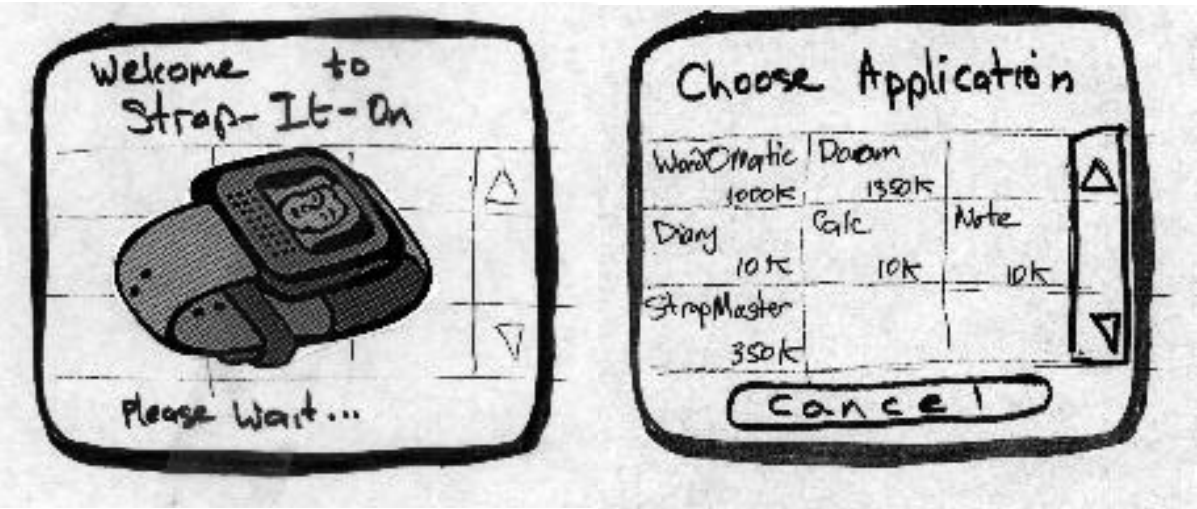
training to understand or manipulate it, while the kiosk administrator could be vitally concerned with memory issues.

The techniques and processes of User Role Modelling from Usage-Centered Design (Constantine & Lockwood, 1999) can be used to identify the different kinds of users a system needs to support, and to characterise the support a system needs to provide to each kind of user.

2. General Purpose Systems. The more general a system's purpose, the more difficult memory allocation becomes. A system may have to support several radically different types of users – say from novices to experts, or from those working on small jobs to those working on big jobs. Even the work of a single user can have different memory requirements depending upon the details of the task performed: formatting and rasterising text for laser printing may have completely different memory requirements to entering the text in the first place. Also, systems may need to run on hardware with varying memory requirements. Often the memory supplied between different models or configurations of the same hardware can vary by a several orders of magnitude; for example the same program might run on systems with 128Kb or systems with 128Mb. The more general a system's purposes, the more leverage there can be in making (or allowing) users to take responsibility for memory use.

Examples

After it has displayed its startup screen, the Strap-It-On wrist mounted PC asks its user to select an application to run. The 'Select an Application' screen also displays the amount of memory each application will need if it is chosen. In this way, the Strap-In-On constantly provides cues that users may need to worry about memory consumption.



❖ ❖ ❖

Specialised Patterns

The rest of this chapter will contain five patterns that present a range of techniques for making the user worry about the systems memory use. They describe ways that a user interface can be structured, how users can be placed directly in control of a system's memory allocation, and how the quality of a user interface can be traded off against its memory use.

FIXED SIZED USER MEMORY describes how user interfaces can be designed with a small number of user memories. Controls to access these memories can be designed directly into the interface of the program, making them quick and easy to access. Fixed sized user memories have the disadvantages that they do not deal well with user data objects of varying sizes, or with more than about twenty memory locations.

VARIABLE SIZED USER MEMORY allows the user to store variable numbers of varying sized data objects, overcoming the major disadvantages of designs based on fixed sized user memory. The resulting interface designs are more complex than those based on fixed sized memory spaces, because users need ways of navigating the storage, and must be careful not to exhaust the capacity.

MEMORY FEEDBACK, in turn, addresses some of the problems of variable sized user memory: by presenting users with feedback describing the state of a system's memory use, they can make better use of the available memory. Memory feedback can also describe the system's use of memory — the amount of memory occupied by application software and system services.

USER MEMORY CONFIGURATION allows users to configure the way systems use memory. Often, information or advice about how a system will be used, or what aspects of a system's performance are most important to its users, can help a system make the best use of the available memory.

Finally, **LOW QUALITY MULTIMEDIA** describes how multimedia resources — a particularly memory-hungry component of many systems — can be reduced in quality or even eliminated altogether, releasing the memory for more important uses in the system.

See Also

The user's memory model may be implemented by **FIXED ALLOCATION** or **VARIABLE ALLOCATION: FIXED SIZE USER MEMORY** is usually implemented by **FIXED ALLOCATION** and **VARIABLE SIZE USER MEMORY** is usually implemented by **VARIABLE ALLOCATION**.

FUNCTIONALITY A LA CARTE [Adams 95] can present the costs and benefits of memory allocations to the user.

A static **MEMORY BUDGET** can provide an alternative to **USER MEMORY CONFIGURATION** that does not require users to manage memory explicitly, but that will have higher memory requirements to provide a given amount of functionality.

The patterns in this chapter describe techniques for designing user interfaces for systems that have limited memory capacity. We have not attempted to address the must wider question of user interface design generally — as this is a topic that deserves a book of its own. Schneiderman's *User Interface Design* is a general introduction to the field of interface design, and Constantine and Lockwoods' *Software for Use* presents a comprehensive methodology for incorporating user interface design into development processes. Both texts discuss interface design for embedded systems and small portable devices as well as for desktop applications.

Fixed Size User Memory

Also known as: Memories

How can you present a small amount of memory to the user?

- You have a small amount of user-visible memory.
- Users need to store a small number of discrete items in the memory
- Every item users need to store is roughly the same size
- Users need to be able to retrieve data from the memory particularly easily.
- Users cannot tolerate much extra complexity in the interface.

Some systems have only a small amount of memory available for storing the users' data (and presumably only a small amount of data that users can store). This user data is often a series of discrete items, such as telephone numbers or configuration settings, where each item is the same size. For example the Strap-It-On needs to definitions for its voice-input feature. Each macro requires enough memory to store both a three second spoken phrase of the user's choice and the commands that are to be executed when the user speaks that phrase.

Users need to be able to retrieve data from memory quickly and easily — after all, that's why the system is going to the trouble to store such a small amount of data. For example the point of the Strap-it-On's voice input macros are to make data entry more efficient, streamlined, and "fun to do all day" (to quote the marketing brochure).

Similarly music synthesisers will store multiple 'patches' (pieces of music) so that they can be quickly recalled during a performance, and telephones store numbers because people want to dial them quickly.

One approach to this problem is to let the user choose things to store until the device is out of memory, when it stops accepting things. This is quite easy to implement but becomes unsatisfactory when there's only a small amount of memory. Users will tend to get the idea that the device has an infinitude of memory, and consequently will be surprised when the system refuses their requests. Also you'll need some kind of interface to retrieve things from the memory, to delete things that have already been stored, and so on — all of which will just take up more precious memory space.

Therefore: *Provide a fixed number of fixed size memory spaces, and let the user manage them individually.*

Ensure the UI design makes it clear that there are a fixed number of user memory spaces — typically by allocating a single interface element (such as a physical or virtual button) for each memory. Ideally each memory space should be accessed directly via its button, or via a sequence number, to reinforce the idea that there are only a fixed number of user memories.

The user can store and retrieve items from a memory space by interacting with the interface element that represents it. Ideally the simplest interaction, such as pressing the button that represents a user memory, should retrieve the contents of the memory — restoring the device to the configuration stored in the memory, running the macro, or dialling the number held in that memory. Storing into a user memory can be a more complex interaction, because storing is performed much less frequently than retrieval. For example, pressing a "store" button and then pressing the

memory button might store the current configuration into that memory. Any other action that uses the memory should access it in the same way.

An interface with a fixed number of user memories does not need to support an explicit delete action from the memories; the user simply chooses which memory to overwrite.

For example, the Strap-It-On allocates enough storage for nine voice macros. This storage is always available – allocated permanently in the **MEMORY BUDGET.** You can access the set-up screen via the Strap-It-On's operating system, and it shows only the nine memory spaces available.

Consequences

The idea that the system has a fixed number of fixed size user memory spaces into which users can store data is obvious from the its design, making the design *easy to learn*. The fixed number of user memory spaces becomes part of users' model of the system, and the amount of memory available is always *clear to users*. Users should never experience the system running out of memory — rather, they will just have to decide which user memory to overwrite.

Since the memory is pre-allocated, the system never needs to produce any error messages explaining that the system has run out of memory; this makes the system more *usable* and *easy to implement*. Because the number of memories is fixed, the interface can be designed so that users *can easily and quickly* choose which memory to retrieve. The *graphical layout* of the interface is made easier, because there are always the same number of memories to display.

However: The user interface architecture is strongly governed by the memory architecture. This technique works well for a small number of memory spaces but *does not scale* well to allocating more than twenty or thirty memory spaces, or storing objects of more than two or three different sizes.

Increasing the number of fixed user memories can require a redesign of the interface, especially if memories are represented in the UI.

❖ ❖ ❖

Implementation

1. Accessing Memories. There are three main interface design techniques that can be used to access fixed size user memories: *direct access, banked access, and sequential access*.

1.1. Direct Access. For a small number of user memories, allocate a single interface element to each memory. With a single button for each memory, pressing the button can recall the memory directly — a fast and easy operation. Unfortunately, this technique is limited to sixteen or so memories because few interfaces can afford to dedicate too many elements to memory access.

For example, the drum machines in the ReBirth-338 software synthesiser provide a fixed size user memory to store drum patterns. The sixteen buttons across the bottom of the picture above

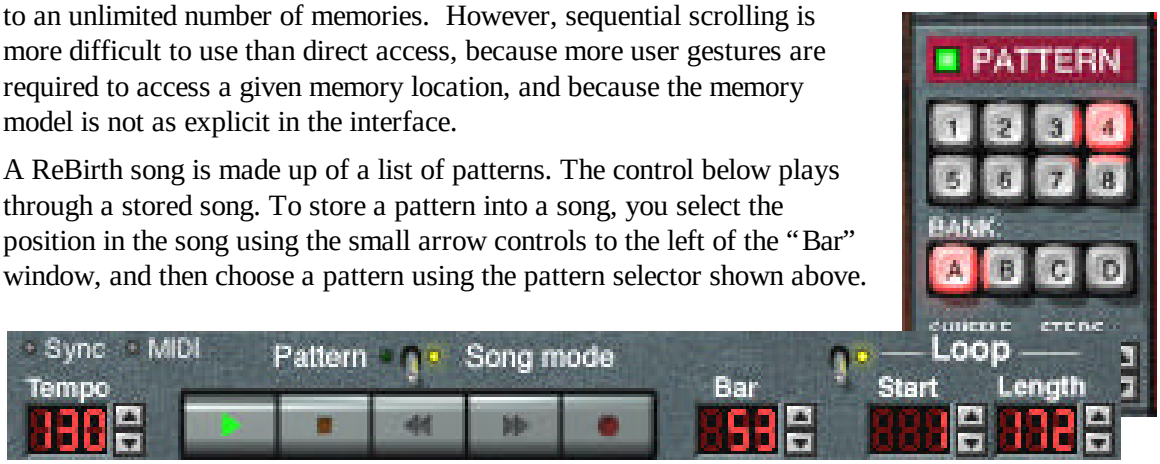
correspond to sixteen memory locations storing sixteen drum beats making up a single pattern — we can see that the bass drum will play on the first, seventh, eleventh and fifteenth beat of the pattern.

1.2 Banked Access. For between ten and one hundred elements, you can use a two dimensional scheme. Two sets of buttons are used to access each memory location — the first set to select a memory bank, and the second set to select an individual memory within the selected bank. Banked access requires fewer interface elements than direct access given the same number of memory spaces, but is still quick to operate.

Again following hardware design, the ReBirth synthesiser can store thirty-two patterns for each instrument in emulates. The patterns are divided up into four banks (A, B, C, D) of eight patterns each, and these patterns are selected using banked access.

1.3 Sequential Access. For even less hardware cost (or screen real estate) you can get by with a couple of buttons to scroll sequentially through all the available memory spaces. This approach is common on cheap or small devices because it has a very low hardware cost and can provide access to an unlimited number of memories. However, sequential scrolling is more difficult to use than direct access, because more user gestures are required to access a given memory location, and because the memory model is not as explicit in the interface.

A ReBirth song is made up of a list of patterns. The control below plays through a stored song. To store a pattern into a song, you select the position in the song using the small arrow controls to the left of the "Bar" window, and then choose a pattern using the pattern selector shown above.



2. Naming Memories. Where there are more than four or five memories you can consider allowing the user to name each memory to make it easier for users to remember what is stored in each memory space. A memory name can be quite short — say eight uppercase characters — and so can be stored in a small amount of memory using simple **STRING COMPRESSION** techniques.

There is still a trade-off between the *memory* requirements for storing the name and the *usability* of a larger number of memories, but there is no point providing a system with large numbers of memories if users can't find the things they have stored in them. If the system includes a larger amount of preset data in stored in **READ-ONLY STORAGE** you can supply names for these memories, while avoiding naming user memories stored in scarce RAM or writeable persistent storage.

3. A Single User Memory Space. Systems that provide just one memory space are special cases of fixed sized user memory. For example, many telephones have a "last number redial" feature that is a single user memory set after every number is dialled. Similarly, many laptop computers have a single memory space for storing backup software configuration parameters, so that the machine can be rebooted if it is misconfigured. A single memory space is usually quick and easy to access, but obviously cannot store much information.

4. Variable Sized Objects. A system with a fixed number of fixed sized memory locations cannot cope well with storing objects of varying sizes. If an item to be stored is too small for the memory

space, the extra space is wasted. If an item is too large, there are several options: to truncate it, to store it in two or more spaces, or to refuse to store it by presenting an error to the user.

If an interface needs to store just two or three kinds of different sized user data objects the interface design can have a separate set of user memories for each kind of object that needs to be stored. However the size and number of the memory spaces must still be determined in advance, and it is unlikely that it will match the number of objects of the appropriate kind that each user wishes to store.

5. Initialising Memories. You may be able to simplify the design and add value for the user by pre-initialising the memory spaces. It's simpler, because there is no need to implement the exceptional handling for uninitialised spaces. For this to work, you need to find good initial contents for the objects to be stored. The memory spaces of synthesisers and drum machines, for example, typically start with useful sounds or drum patterns than can be used immediately and later overwritten.

One compensating advantage of implementing the idea of empty memories is that you can support an *implicit store* operation, which stores an object into some empty memory space without the user having to choose the space explicitly. This certainly makes the store operation easier, but an implicit store operation can fail due to lack of memory — effectively treating the fixed size memories as if they were variable sized. The Nokia 2210e mobile phone supports implicit stores into its internal phone book, but if there are no empty memories users can choose which memory to overwrite.

Example

The StrapItOn PC has nine memories that can be used to store voice input macros. Each memory is represented by one of nine large touch-sensitive areas on the StrapItOn screen. To record a macro, users touch the screen area representing the memory where they want to store the macro — if this memory is already in use, the existing macro is overwritten.

The Korg WaveStation synthesiser contains several examples of fixed sized user memories. The most obviously is that its user interface includes five soft keys that can be programmed by the user to move to a particular page in the WaveStation menu hierarchy. The soft keys are accessed by pressing a dedicated "jump" key on the front panel, and then one of the five physical keys under the display.

The main memory architecture of the WaveStation is also based on fixed sized user memories. Each WaveStation can store fifty 'performances', which can each refer to four of thirty-five 'patches', which can each play one of thirty-two 'wave sequences'. Patches and performances have different sizes and are stored in their own fixed-sized user memories. If you run out of patch storage, you cannot use empty performance memories. Users address the patch and performance memories explicitly with memory location numbers entered by cursor keys, by turning a data-entry

knob, or by using a numeric keypad. Patches, performances and wave sequences can be named — performances and patches with up to fifteen characters, wave sequences only up to seven.

❖ ❖ ❖

Known Uses

GSM mobile phone SIM (Subscriber Information Module) cards are smart cards that store a fixed number of phone memories each containing a name and a number. The number of memories depends on the type of SIM. Users access the memories by number. The same SIM cards can also store a fixed number of received SMS (short message service) text messages; an incoming SMS is implicitly stored, or the user is told if the store overflowed.

An early Australian laser printer required the user to uncompress typefaces into one of a fixed number of memory locations. For example, making Times Roman, Times Roman Italic, and Times Roman Bold typefaces available for printing would require three memory locations into which the ROM Times Roman bitmaps would be uncompressed, with some bitmap manipulations to get italic and bold effects. Documents selected typefaces using escape codes referring to memory locations. Larger fonts had to be stored into two or more contiguous locations, making the user worry about memory fragmentation as well as memory requirements, and giving very interesting results if an escape code tried to print from the second half of a font stored in two locations.

The Ensoniq Mirage sound sampler was the ultimate example of making the user worry. The poor user — usually a musician with little computer experience — had to allocate memory for sound sample storage by entering two digit hexadecimal numbers using only increment and decrement buttons. Each 64K memory bank could hold up to eight samples, provided each sample was stored in a single contiguous memory block. In spite of the arcane and frustrating user interface (or perhaps because of the high functionality the interface supported with limited and cheap hardware) the Mirage was used very widely by mid-1980s popular musicians, and maintains a loyal if eccentric following ten years later.

See Also

VARIABLE-SIZED USER MEMORY offers an alternative to this pattern that explicitly models a reservoir of memory in the system, and allows users to store varying numbers of variable sized objects.

MEMORY FEEDBACK can be used to show users which memories are empty and which are full, or provide statistics on the overall use of memory (such as the percentage of memories used or free).

Although systems' requirements do not generally specify **FIXED SIZED USER MEMORIES**, by negotiating with your clients you may be able to arrange a **FEATURECTOMY**.

A **MEMORY BUDGET** can help you to design the number and sizes of **FIXED SIZED USER MEMORIES** your system will support.

You can use **FIXED ALLOCATION** to implement fixed sized user memories.

Variable-Sized User Memory

How can you present a medium amount of memory to the user?

- You have a medium to large amount of user-visible memory
- Users need to store a varying number of items in the memory
- The items users can store vary in size
- The memory requirements for what the user will need to store are unpredictable.

Some programs have medium or large amounts of memory available for storing user data. For example the Strap-It-On wrist-portable PC provides a file system to allow users to store application data. Files can vary in size from a few words to several pages, some users may choose to store a few large files while other users may store many small files. The behaviour of some users changes over time: one week storing many small files, the next one large file, the next a mixture.

One approach to organising the memory would be to provide **FIXED SIZED USER MEMORY**. The system could allocate a fixed number of fixed-sized spaces to hold memos the user wishes to store. Of course, this suffers from all the problems of fixed allocation: memory spaces holding small memos will waste the rest of the space, and long memos must somehow be split over a number of different spaces. Another alternative would be to require users to pre-allocate space to store memos, but this requires users to be able to accurately estimate the size of a new memo before it is created, and the pre-allocation step will greatly complicate the user interface.

Therefore: *Use variable allocation, but make the user aware of how the storage used by each item impacts on the storage available.*

Allow the user to store and retrieve items flexibly from the systems' memory reservoir. The reservoir does not have to be made explicit in the interface design — although it may be. Each item stored in the memory should be treated as an individual object in the user interface, so that users can manipulate it directly. You also need to provide an interface to allow the user to find particular items they want to use, and to explicitly delete objects from the reservoir making the memory space available for the storage of new objects.

For example the Strap-It-On uses **VARIABLE SIZED USER MEMORY** for its file system. It has reservoir large enough to hold ten thousand words (about a hundred thousand characters of storage) to hold all the users' files. When users create new files, these are stored within the reservoir until they are explicitly deleted. The file tool displays the memory used by every file in a browser view, the percentage of free memory left in the reservoir pool in its status line, and also uses error messages to warn the user when memory use exceeds certain thresholds (90%, 95% 99%).

Consequences

Users can store new objects in memory quite *easily*, provided there is enough space for them. Users can make *flexible* use of the main memory space, storing varying numbers and sizes of items to make the most of the systems capacity — effectively reducing the system's *memory requirements*. Users don't need to choose particular locations in which to store items or to worry about accidentally overwriting items or to do pre-allocation, and this increases the system's *usability*.

Variable sized memory allocation is generally quite *scalable*, as it is easier to increase the size of a reservoir (or add multiple separate reservoirs) than to increase the number of fixed size memories.

However: The program's *memory model is exposed*. The user needs to be aware of the reservoir, even though the reservoir may not be explicitly presented in the interface. The user interface will be *more complex* as a result, and the graphical design will be more difficult. Users will not always be aware of how much memory is left in the system, so they are more likely to *run out of memory*.

Any kind of variable allocation decreases the *predictability* of the program's memory use, and increases the possibility of *memory fragmentation*. Variable sized allocation also has a higher *testing cost* than fixed sized allocation.

❖ ❖ ❖

Implementation

Although a **VARIABLE SIZED USER MEMORY** can give users an illusion of infinite memory, memory management issues must still lurk under this façade: somewhere the system needs to record that the objects are all occupying space from the same memory reservoir, and that the reservoir is finite. Even if the reservoir is not explicit in the interface design, try to integrate memory use with the rest of the system and the domain model, so that the user can understand how the memory management works. Consider using **MEMORY FEEDBACK** to keep the user informed about the amount of memory used (and more importantly, available) in the reservoir.

A user interface based on **VARIABLE SIZED USER MEMORY** is more sophisticated than a similar model built on fixed sized user memory. A model of variable sized user memory must include not only empty or full memory locations, but also a more abstract concept of "memory reservoir" that can be allocated between newly created objects and existing objects if their size increases. The objects that can be stored in user memory can also be more sophisticated, with varying sizes and types.

Here are some further issues to consider when designing user interfaces that provide variable user memory spaces:

1. Multiple Reservoirs. You can implement multiple reservoirs to model multiple hardware resources, such as disks, flash ram cards, and so on. Treat each separate physical store as an individual reservoir. You need to present information about each reservoir individually, as the amount and percentages of free and used memory. You can also provide operations that work on whole reservoirs, such as backing up all the objects from one reservoir into another or deleting all the objects stored in a reservoir.

You will also need to ensure that the user interface associates each object with the reservoir where it is stored — typically by using reservoirs to structure the way information about the objects is presented, by grouping all the objects in a reservoir together. For example most desktop GUI filing systems show all the files in a single disk or directory together in one window, so that the association between objects and the physical device on which they are stored is always clear.

2. Caching and Temporary Storage. Caching can play havoc with the user's model of the memory space if applications trespass on that memory for caches or temporary storage. For example many web browsers on palmtop and desktop machines maintain temporary caches in user file storage. The minor problem here is that naïve measurements of the amount of free space will be too low, as some of the space is allocated to caches; the major problem is that unless the memory is somehow released from the caches it cannot be used for application data storage.

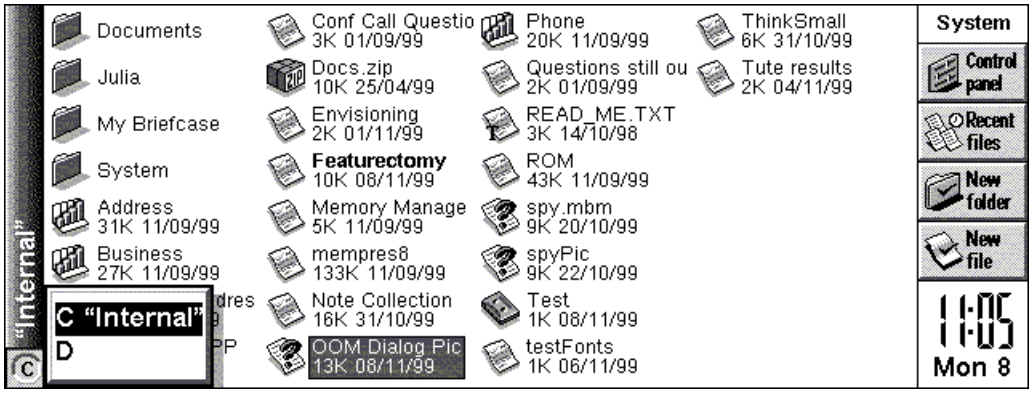
The **CAPTAIN OATES** pattern describes how you can design a system so that applications release their temporary storage when it is required for more important or permanent uses. The key to this pattern is that when memory is low, the system should signal applications that may be holding cached data. In response to receiving the signal, any applications holding cached data should release the caches.

3. Fragmentation. As with any kind of **VARIABLE ALLOCATION**, reservoirs may be subject to fragmentation. This can cause problems as the amount of memory reported available may be less than the amount of memory that can be used in practice. So for example, while the Strap-It-On's file memory may have 50K free characters, the largest single block might be only 10K – not enough to create a 15K email message.

One way to avoid this problem is to show users information about the largest free block of memory in each reservoir, rather than simply the amount of free memory. But this approach complicates the users' conceptual model of the system. Other approaches are to implement automatic **MEMORY COMPACTION**, or to choose data structures that do not require explicit compaction

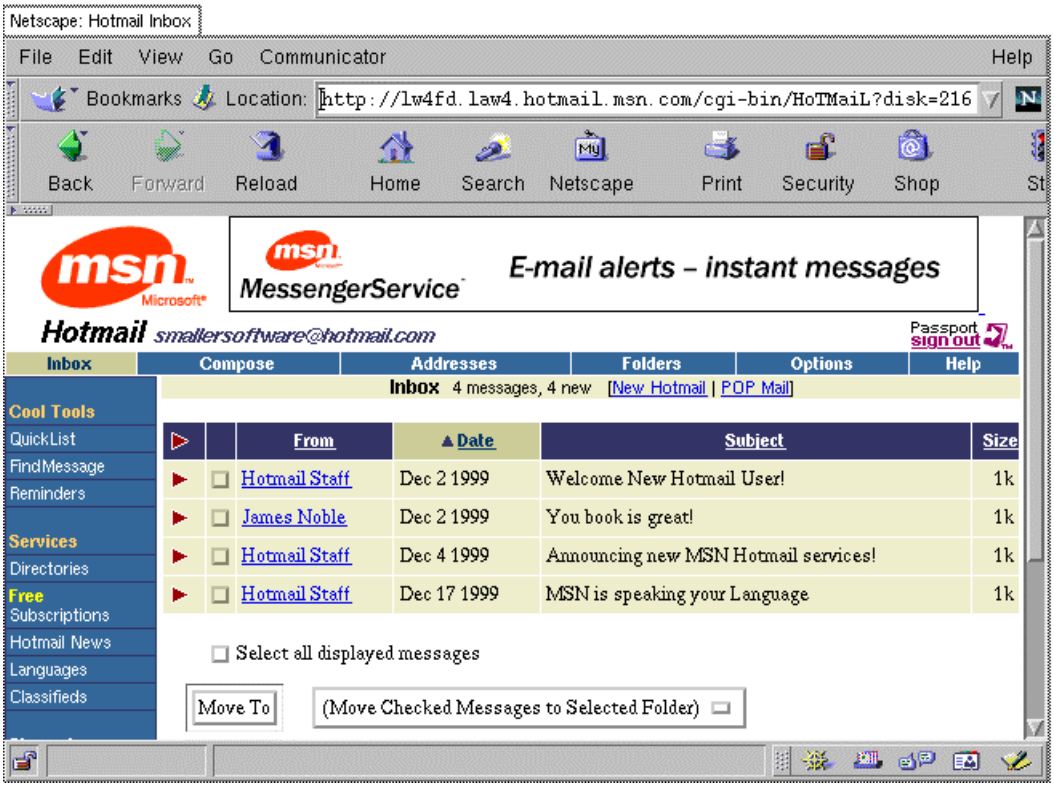
Examples

A Psion Series 5 allows users to store text files and spreadsheet files in a persistent storage reservoir (called a 'drive' but implemented by battery backed up RAM). The browser interface in the figure below shows the files stored in a given drive, and the size of each file. Clicking on the 'drive letter' in the bottom left hand corner of the screen produces a menu allowing users to view a different reservoir.



The StrapItOn PC can also list all the files in its internal memory, and also always lists their sizes.

The Hotmail web-based mail server also uses variable sized user memory. When Hotmail lists the messages received by a mail account, it also lists the size of each message, tucked to the far right-hand-side of the screen.



Hotmail may seem like a strange example of a 'small system', but Hotmail must support hundreds of thousands of separate mail accounts. Because this requires a very large total amount of memory, Hotmail limits the amount of memory required by each individual account, and makes the user worry about the amount of memory their mail occupies. If an account occupies too much memory, incoming messages are returned to sender.

❖ ❖ ❖

Known Uses

Most general purpose computers provide some kind of variable sized user memory for storing users' data — either in a special region of (persistent) primary storage, such as the PalmPilot, Newton, and Psion Series 5, or on secondary storage, such as PCs, workstations, minis and mainframes. Similarly, multitasking computers effectively use **VARIABLE SIZED USER MEMORY** — users can run applications of varying sizes until they run out of memory.

See also

Use **VARIABLE ALLOCATION** to implement variable sized user memory. **MEMORY FEEDBACK** can help the user avoid running out of memory. The size of the reservoir may be able to be set by **USER MEMORY CONFIGURATION. FIXED SIZED USER MEMORY** can be an alternative to this pattern if only a few fixed sized objects need to be stored.
