Thinking Small
The Processes for Creating Small Memory Software

Abstract:

This paper describes some process patterns for teams to follow when creating software to run in limited memory.

It is a draft version of a chapter to add to the authors' book Small Memory Software, and follows the structure of other chapters in that book.

Major Technique: Thinking Small

A.k.a Small methodology, 'Real' programming.

How should you approach a small system?

- You're developing a system that will be memory-constrained.
- There are many competing constraints to satisfy
- If different developers take different views on which things to optimise, they will produce an inconsistent system that satisfies *none* of the constraints.

You're working on a project and you suspect there will be resource limitations in the target system. For example, the developers of the 'Super-spy 007' version for the Strap-it-On wrist-mounted computer face a system with only 200 Kb of RAM and 2 Mb ROM. How are they to adjudicate the demands of the voice recognition software, the vocabularies and the software-based radio, to make it a secret agent's dream toy? Should they store the vocabularies in ROM to save RAM space, or keep them in RAM to allow them to change from Russian to Arabic on the fly? What, in short, is important in their system, and what is less so?

In many projects it's clear from the outset that the development team will have to spend at least some time and effort satisfying the system's memory limitations. You have to cut your coat to fit your cloth. Yet if the team just spends lots of effort optimising everything to work in very limited memory, they'll waste a lot of time or maybe produce a product that could have been much better. Worse still the product may fail to work at all because they have been optimising the wrong thing.

For example, any of the following facilities may be limited:

- Heap (RAM) space for the whole system
- Heap space for individual processes (if the maximum heap size of a process is fixed)
- Process stack size
- Secondary storage use
- ROM space (for programs that execute from ROM)

Optimising one of these will often be at a cost from one of the others. In addition techniques that optimise memory use will tend to compromise time-performance, usability or both.

In any system the architects will have to moderate the demands of different components in the system against each other. That is a big and highly sensitive task.

Software programmers tend to take their design decisions seriously, so capricious decisions can cause friction or worse within a development team.

You might hope to use clever techniques to defer the key decisions about these priorities until later in the project, when you'll know more about the implementation. But in practice many of the most important strategic decisions cannot be deferred, as they pervade the entire system and provide a framework for later decisions. Such strategic decisions are reflected, for example, in the interfaces between components, in the trade-off between ROM and RAM, and in the question of whether or not to use Partial Failure in components.

Design decisions about the trade-offs based on just individual designers' foibles, on gut feel or on who shouts loudest will lead neither to consistent successful designs, nor to a harmonious development. You'll need a more objective approach.

Therefore: *Devise a memory strategy for the entire project.*

First draw up a crude **MEMORY BUDGET** of the likely available resources in each of the categories above. If the figures are flexible (for example, if the system is to run on standard PCs with variable configurations and other applications), then estimate or negotiate target values with clients. Meanwhile, also estimate very approximately the likely memory needs of the system you're developing. Identify the tensions between the two. Identify the design decisions that will significantly challenge the memory use, and ensure these decisions happen early.

Based on this comparison you'll be in a position to identify which constraints are most vital. It may be a constraint on one of the forms of memory in the system. Other constraints – time constraints, reliability, usability – may also be as or more important.

Enshrine these priorities as a core 'given' for everyone working on the project. Ensure that absolutely everyone working on the team understands the priorities. Write the strategy in a document; make presentations; distribute the T-shirt! Indoctrinate each new developer who joins the team afterwards with the same priorities.

Once you've identified your priorities, you'll be in a position to plan how to approach the rest of the project. You may need a formal **MEMORY BUDGET**, or perhaps **MEMORY TRACKING**. Or you may choose to leave **MEMORY OPTIMISATION** until near the end of the project. Depending on the nature of the system, you may need to plan for **EXHAUSTION TESTING**, or assign time to **PLUG THE LEAKS**.

For example, the developers of the 'Super-spy 007' decided the important priority was the constraint on RAM, since RAM memory provided the only storage – and a reset might then erase vital information about the Master Villain's plans to destroy the world! The next priority was user response (to give a quick response in dangerous situations). So the components and interfaces are designed to minimise this memory use, and then to give reasonable user response.

Consequences

Every member of the team will understand the priorities. Individual designers will be able to make their own decisions knowing that the decisions will fit within the wider context of the project. Design decisions by different teams will be consistent, adding to the coherence of the system developed.

You can estimate the impact of the memory constraints on project timescales, reducing the uncertainty of the project.

The initial estimates of memory needs can provide a basis for a more formal **MEMORY BUDGET** for the project.

However: Deciding the strategy takes time and effort at an important stage of a project.

Sometimes later design decisions, functionality changes, or hardware modifications may modify the strategy; this invalidates the earlier design decisions, so might leave the project in a worse position than if individuals had taken random decisions.

Implementation Notes

Sometimes it's not necessary to make the strategy explicit. Many projects work in a well-understood context. For example an MS-Windows 'shrink-wrapped' application can assume a total system size of more than 12Mb RAM (and more than 30Mb paged memory), about 50Mb disk and program space – as we can deduce by studying any number of 'industry standard' applications.

So MS Windows developers share an unwritten understanding of the memory requirements of a typical program. The strategy of all these Windows applications and the trade-offs will tend to be similar, and these are often encapsulated in the libraries and development environments or in the standard literature. Given this 'implicit strategy' it may be less necessary to define an explicit one; any programmer who has worked on a similar project or read up the literature will unconsciously choose appropriate trade-offs.

However having an implicit strategy for all applications can cause designers and programmers to overlook lesser but still significant variations in a specific project. For example a Windows photograph editor will randomly access large amounts of memory. So it may have to assume (and explicitly demand) rather more real, rather than paged, memory than other 'standard' applications.

Developers from Different Environments

Programmers and designers used to one strategy often have very great difficulty changing to a different one. For example, many MS Windows programmers coming to the EPOC or Palm operating systems have great difficulty internalising the idea that programs must run indefinitely even if there's a possibility of running out of memory. Windows CE developers have even more of a problem with this, as the environment is superficially similar to normal Windows.

Yet if such programmers continue to program in their former 'large workstation' paradigms, the resulting code has poor quality, and often doesn't satisfy user needs. The developers need to adapt to the new strategies.

One excellent way to promote such 'Thinking Small' is to exaggerate the problem. Emphasise the smallness of the system. Make all the developers imagine the system is smaller than it is! And encourage every team member to keep a very tight control on the memory use. Ensure that each programmer knows which coding techniques are efficient in terms of memory, and which are wasteful. You can use design and code reviews to exorcise wasteful features, habits and techniques.

In this way you can develop a culture where memory saving is a habit. Wonderful!

Guidelines for Small System Development

The following are some principles for designing memory limited software:

Design small, code small	You need to build in memory saving into the design as well as into the code of individual components. The design provides much more scope for memory saving

	than code.
Create bounds	Avoid unbounded memory use. Unlimited recursion, or algorithms without a limit on their memory use, will almost certainly eventually cause irritating or fatal system defects.
Design for the default case	It's always tempting to design your standard object data structures to handle every possible case. But this approach tends to waste memory. It's better to design objects so that their default data structure handles only the simplest case, and have extension objects [Beck ?] to handle special cases.
Minimise lifetimes	Heap- and stack- based objects cease to take up memory when they're deleted. You can save significant memory by ensuring that this happens as early as possible [KenA list in Cacheable Expression]

Extreme vs. Traditional Projects

There are many different styles for teams working on software development. To highlight some of the differences, we'll contrast two opposing styles: 'Traditional Development' and 'Extreme Programming'.

Traditional development [Gilb], [DeMarco] derives its processes and targets from the project controlling techniques used successfully in other engineering disciplines. Each developer is responsible for there own areas of code. A project starts with the team agreeing or receiving a set of specifications from clients at the start of the project – typically as a Functional Specification document. If the project is large enough, a design team will next decide on the architecture and specify the software components for the system. Then separate teams will work on written designs for each component and for the interfaces between them. Finally each team works separately on implementing each component, usually with each programmer responsible for a section of the code and functionality. Either the component programmers or a different team will be responsible for component testing, and then for system testing. Finally the system is 'released' and shipped to the customer, followed by either new projects to modify the functionality, or 'maintenance' to fix defects and shortcomings as required.

In the 'Extreme Programming' style of development, there is a single development team of up to about twelve programmers. Development works in short cycles of a week or so, each cycle culminating in a release – which may potentially be shipped to a customer. The team interacts strongly with their customer to develop only the most important features in each cycle. Programmers always work in pairs, develop complete test code before any implementation, have a strong emphasis on 'refactoring' existing code to satisfy new requirements, and eschew formal design documentation.

One might compare the two approaches to two different ways of house building. A property speculator will create a building by hiring a number of professionals, and arranging for the design to be done first, the builders to ready at the right time to start, the plumbers to be available when the builders have finished the shell, etc.

Extreme programming is more like a couple building their own house. They create the shell, make one room liveable, and take on new projects to improve their facilities and add new rooms when time and money permit.

[Insert here. How you use the patterns in a traditional project. How you use the patterns in an Extreme Project. Get feedback from Kent Beck.]

In a traditional project the architectural strategy is a part of the architect's Vision [?ref. JD?].

In an XP project, the strategy will best be reflected in the project 'metaphor'. [XP ?Wiki]. Individual memory constraints are reflected as 'stories', which become test cases that every future system enhancement must support.

Specialised Patterns

The rest of this chapter introduces six further 'process patterns' commonly used in organising projects with limited memory. Process patterns differ from design patterns in that they describe what you do – the process you go through – rather than the end result.

These patterns apply to virtually all small memory projects, from one-person developments to vast systems involving many teams of developers scattered worldwide. Throughout this chapter we'll use the phrase 'development teams' to mean 'all of the people working on the project'. If you're working alone, you should read this as referring to yourself alone; if a single team, then it refers to just that team; if a large project, it refers to all the teams.

Equally, the patterns themselves work at various levels of a project's organisation. Suppose you're working on the implementation of the Strap-It-OnTM wristwatch computer. The overall project designers ('system architecture team') will use each pattern to examine the interworking of all the components in the system. Each separate development team can use the patterns to control their implementation of their specific component, working within the parameters and constraints defined by the system architecture team.

The patterns are as follows:

Memory Budget How do you keep control in a project where memory is very tight? Draw up a memory budget, and plan the memory use of each component in the system.

Featurectomy How do you ensure you have an implementable set of system requirements given the system restraints? Negotiate with the clients, users and requirements specification teams to produce a specification to satisfy both users needs and the system's memory constraints.

Memory Tracking How do you find out if the implementation you're working on will satisfy your memory requirements? Track the memory use of each release of the system, and ensure that every developer is aware of the current score

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

Plugging the Leaks How do you ensure your program recycles memory efficiently? Test your system for memory leakage and fix the leaks.

Exhaustion Test How do you ensure that your programs work correctly in out of memory conditions? Use testing techniques that simulate memory exhaustion.

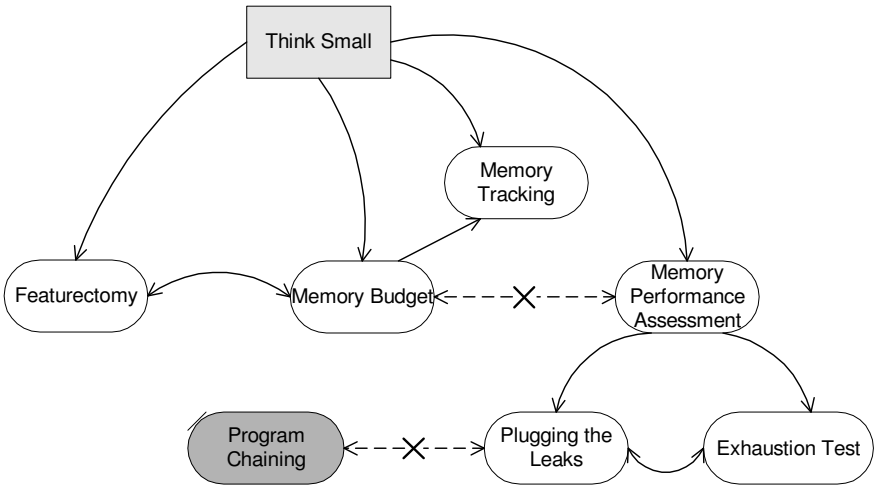

Figure 1: Process Pattern Relationships

Known Uses

The EPOC operating system is ported to many different telephone hardware platforms; each has a different configuration of ROM, RAM and Flash (persistent) memory. So each environment has a different trade-off and application strategy. Some have virtually unlimited non-persistent RAM; others (such as the Psion Series 5) use their RAM for persistent storage so must be extremely parsimonious with it.

In each case, the memory strategy is reflected in the choice of **Data Structures,** in **User Interfaces,** and in the use of **Secondary Storage**. The Psion Series 5 development used an implicit strategy, passed by word of mouth. Later ports have had an explicit strategy documents.

See Also

THINKING SMALL provides a starting point for a project. Most of the other patterns in this book have trade-offs that we can evaluate only in the context of a memory strategy.

Memory Budget Pattern

A.k.a. Memory Costings

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

- You're doing a project where memory is limited and there's a risk that the project will fail if its memory requirements exceed these limits.
- You have several different components or tasks using memory
- Different individuals or teams may be responsible for each.
- Saving memory costs effort – better let someone else do it!
- Unnecessary optimisation would waste programmer time.

You are working on a software development project, and you've identified that there's a possibility that memory constraints may be a significant problem.

For example, the whole Strap-It-On project is obviously limited by memory from the beginning. The Strap-It-On needs to be as small, as cheap, and as low-powered as possible, but also be usable by computer novices and have enough capacity to be adopted and recommended by experts.

If you don't take sufficient care of the memory constraints in the system design and implementation, bad things will happen to the project. Perhaps the system will fail to work at all; perhaps users will get inadequate performance or functionality; or perhaps the cost of the extra memory hardware will make the software unsaleable.

You could have everyone involved design and code so as to reduce their memory requirements to the bare minimum. That would certainly reduce the risk of the system becoming too big. But there are be costs to this scorched earth approach – if you concentrate on keeping memory low, then you'll have to accept trade-offs elsewhere such as poor time performance, difficult-to-use interfaces or large amounts of developer effort. It would be poor engineering to concentrate on one aspect, memory use, to the exclusion to all others. More importantly, how can you decide what the "bare minimum" actually is? You could save all the memory by deciding not to implement the program!

In almost any modern system you will be developing or using several components, each with its own memory requirements. Some will provide more opportunities for memory saving than others. There's no point in working overtime to save a few bytes in one component, when a minor change in another would save many times that. How do you decide which components to concentrate on?

In many projects there will be several teams each working on different components. Each individual team may feel they have less incentive to save memory than other teams — everyone likes to believe that the problem they are working on is unique, and harder than everyone else's problem. It costs teams programmer effort to reduce memory use – so they'll be tempted to let a different team pay the cost, treating memory as "someone else's problem". How can you share out the pain of saving memory between the teams, so that they can design their software and plan its implementation effectively?

Therefore: *Draw up a memory budget, and plan the memory use of each component in the system.*

Define memory consumption targets for the each component as part of the specification process. Ensure that the targets are measurable [Gilb88], so the developers will be able to check whether they're within budget.

This process is similar to the 'costings' process preceding any major building work. Surveyors estimate costs and time of each part of the process, to determine the feasibility and to negotiate the requirements of the customer.

Ensure that all the teams 'buy into' the budget. Involve them in the process of deciding the figures to budget, estimating the memory requirements and negotiating how any deficits are split between the different teams. Communicate the resulting budget to all the team members and invite their comments. Refer to it while doing **MEMORY TRACKING** during development, and during the **MEMORY PERFORMANCE ASSESSMENT** later in the project. Make meeting the budget a criterion for release of each component. Celebrate when the targets are met!

Consequences

The task of setting and negotiating the limits in the memory budget encourages all the teams to **THINK SMALL**, and sets suitable parameters for the design of each component. The budget forces the team to take an overall view of memory use, increasing the *architectural consistency* of the system. Furthermore, having specific targets for memory use greatly increases the *predictability* of the memory use of the resulting program, and can also reduce the program's absolute *memory requirements*.

Because developers face specific targets, they can make decisions *locally* where there are trade-offs between memory use and other constraints. It's also easy to identify problem areas, and to see which modules are keeping their requirements reasonable, so a budget increases *programmer discipline*.

However: defining, negotiating and managing the budgets requires significant *programmer effort*.

Developers may be tempted to 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 support for testing memory budget requires *hardware or operating system support.*

Setting fixed memory budgets can make it more difficult to take advantage of more memory if it should become available, reducing the *scalability* of the program.

Formal memory budgets can be unpopular with both programmers and managers because the process adds accountability without direct benefits. If the final system turns out over budget then everyone will loose out; if it turns out under budget then the budget will have been 'wrong' – so those doing the budget may loose credibility.

Implementation Notes

Suiting Budget to the Project

Producing and tracking an accurate memory budget for a large system is a large amount of work, and can impose a substantial overhead on even a small project. If memory constraints aren't actually a problem, maintaining budgets is rather a waste of effort that could be better spent elsewhere. And in an informal environment, with less emphasis on up-front design, developers can be actively hostile to a full-scale formal memory budget.

For this reason, many practical memory budgets are just back-of-the envelope calculations – a few minutes work with the team on the whiteboard, summarised as a paragraph in the design documentation. Only if simple calculations suggest that memory will be tight – or tight in certain circumstances – is it worth spending the effort to put together a more formal memory budget.

What to budget?

There are various kinds of memory use; different environments will have different constraints on each. Here are some possibilities:

- RAM memory usage – heap memory, stack memory, system overheads.
- Total memory usage – including memory **PAGED OUT** to disk.
- ROM use – for systems with code and data in ROM
- Secondary storage – disk, flash and similar data storage, network etc.

In addition, the target environment may add further limitations: a limit on each separate process (such as for code using the 'Small', 16-bit addressing model on Intel architectures), or a limit on stack size (imposed by the operating system).

It's worth considering each constraint in turn, if only to rule most of them out as problems. Often only one or two kinds of memory will be limited enough to cause you problems, and you can concentrate on those.

Dealing with Variable Usage

It's easier to budget ROM usage than RAM. ROM allocation is constant, so you can budget a single figure for each component. Adding these figures together will give the total ROM use for the system.

In contrast, the RAM (and secondary storage) requirements of each component will normally vary with time – unless a component uses only Fixed Data Structures.

One approach is to estimate the worst case memory use of each component and adding the values together, but the result could well be far too pessimistic; in many systems only a few components will be being used heavily at a time. A workstation, for example, will have only a few applications running at any one time – and typically only one or two actively in use.

Yet the memory use of the different components tends not to be independent. For example, if you have an application making heavy use of a, then the applications peak memory usage is likely to coincide with peak memory use in the network driver. How do you deal with this correlation?

To deal with these dependencies, you can identify a number of worst case scenarios for memory use, and construct a budget for the memory use of each component in each scenario. Often, it is enough to estimate an average and a peak memory requirement for each component and then estimate which components are likely to have peak usage for each worst-case scenario. You can then sum the likely use for each scenario; and negotiate a budget so that this sum is less than the total for every one of the scenarios.

Dealing with Uncertainty: Memory Overdraft

Software development in the real world is unpredictable. There's always a possibility for any component that it will turn out to be just too difficult or too expensive in time or other trade-offs to reduce its memory requirements to the budgeted limits. If there are many components, there'll be a good chance that at least

one will be over budget, and the second law of thermodynamics [Flanders&Swan] says it is unlikely that components will be correspondingly under budget.

The answer is to ensure that there is some slack in the budget – an overdraft fund. The amount depends on how uncertain the initial estimates are. Typical amounts might be between 5% and 20%. The resulting budget will be more resilient in the face of development realities, increasing the overall *predictability* of the program's memory use. However you must be careful to ensure that programmers don't reduce their *discipline* and take the overdraft for granted, reducing the integrity of the budget.

The OS/360 project included overdrafts as part of their budgets [Brooks75].

Dealing with Uncertainty: A Heuristic Approach

Having a Memory Overdraft to allocate to defaulting components is a good ad-hoc approach to dealing with uncertainty, but it's a bit arbitrary. If you're seriously strapped for memory, allocating an arbitrary amount to a contingency fund isn't exactly a very scientific approach. Should you assign 5% or 30%? If you assign 30%, you're wasting a very large amount of memory that you could more economically assign to a component. If you assign less, how much are you increasing the risk?

The solution is to use a technique publicised as part of the 'Program Evaluation and Review Technique' (PERT). This is normally used to add together time estimates for project management – see [Filipovitch96], but the underlying statistics work equally well for adding together any set of estimated values.

Make three estimates for each figure rather than just one. Estimate a reasonable worst case value, the most likely (median) value, and a reasonable best achievable (i.e. lowest) maximum value. Try to do your estimation impartially so that it's equally likely that each final figure will turn out higher or lower than the median you've estimated. So when you add them together, probably some of the final figures will be higher and some of them will be lower. In effect combining the all the uncertain figures means that some of the uncertainty 'cancels out'.

The arithmetic of this is as follows. If the estimated value for component i is e_i, and the maximum and minimum values are a_i and b_I, , then the best guess, or 'weighted mean' value for each is:

$$(a_i + 4e_i + b_i) / 6$$

And the best guess of the standard deviation of each is:

$$\sigma_i = (b_i - a_i) / 6$$

So the best estimate of the sum is the sum of all the weighted means; and we calculate the standard deviation of the sum, S_I using:

$$S_i \text{ ^ } 2 = \text{Sum}_i(\sigma_i \text{ ^ } 2)$$

These calculations are very easy to do with a spreadsheet.

For example, Table 1 shows one possible worst-case scenario for the Ring Clock, a kind of watch device worn on the finger than receives radio time checks from transmitters in Frankfurt. This scenario shows it ringing an alarm. Only the Screen Driver and the UI Implementation components are heavily used:

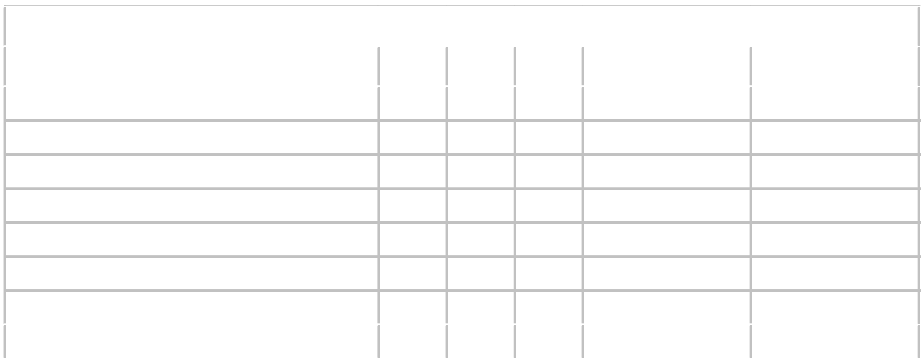

Table 1 : Calculation of the Combination of Several Estimates

A good estimate of the maximum and minimum values for the sum is three standard deviations (the so-called '95% confidence limits') above and below the mean. The table above shows the standard deviation to be roughly 2Kb, which gives the following values for the combined estimates:

Maximum:	41K
Estimate:	35K
Minimum:	29K

So we can be reasonably confident that we shall be able to support this particular worst-case scenario with 41K of memory – much less than the sum of the all the maximum estimates.

If the actual memory available is less, then we might need to work on the most variable estimates (UI Implementation) to produce a more accurate estimate – since the large maximum has contributed much of the uncertainty in the figure. Alternatively we might need to do some **FEATURECTOMY** to reduce the estimated memory requirements of that or other components.

Enforcing the Budget in Software

Some environments provide memory use monitors or resource limits, which you can use to enforce memory budgets. For example IBM UNIX allows you to define a limit on the heap memory of a process, and EPOC's C++ environment can enforce a maximum limit on application heap sizes. The **MEMORY LIMIT** pattern describes how you can implement these limits yourself.

You can use these monitors to enforce the limits on the maximum memory use of each component. Some projects may use these limits for testing only; in other cases they may remain in the runtime system, so that processes or applications will fail (**PARTIAL FAILURE**, or complete failure) rather than exceed their budget.

Of course software limits enforce only the maximum use for each component. Typical worst case scenarios will have only a few components using their maximum memory requirements, so such software limits don't provide a full check that the components are conforming to the budgets.

Example

The Palm Pilot has an interesting budget for its dynamic heap (used for all non-persistent data). Because only one application runs at a time (**PROGRAM CHAINING**), the budget is the same for every application that can run on a given machine.

The following is the Pilot's budget for PalmOs 3.0, for any unit with more than 1 Mbyte of memory [PalmBudget]. Machines with less memory are even more constrained.

24k	System globals (screen buffer, UI globals, database references, etc.)
32k	TCP/IP stack, when active
Variable amount	IrDA stack, "Find" window, other system services
4k (by default)	Application stack (the application can override this amount)
up to 36k	Available for application globals, static data, dynamic allocations, etc.

Table 2: Palm Pilot 3.0 Memory Budget

Known Uses

[Brooks75] discusses the memory budget for the OS/360 project. In that project, the managers found it important to budget for the total size of each module (to prevent paging), and to specify the functionality required of each module as a part of the budgeting process (to prevent programmers from offloading functionality onto other components).

A current mobile phone project has two particular architectural challenges provided by a hardware architecture originally defined for a very different software environment. First, ROM (flash RAM) is very limited. Based on a Memory Budget, the team devised compression and sharing techniques, and negotiated Featurectomy with their clients.

Secondly, though RAM in this phone is relatively abundant, restrictions in the memory management architecture means that each process must have a pre-allocated heap, so every process uses the RAM allocated to it at all times. Thus the team could express the RAM budget in terms of a single figure for each process – the maximum, or worst case, figure.

The Palm documentation specifies a standard memory budget for all Pilot applications. Since only one application runs at a time, this is straightforward.

See Also

There are three complementary approaches to developing a project with restricted memory. The **MEMORY BUDGET** pattern describes how to tackle the problem up front, by predicting limits for memory, and then implementing the software to keep within these limits. The **MEMORY TRACKING** pattern gathers memory use statistics from developers as the program is being built, encouraging the developers to limit the contribution of each component. Finally, if memory problems are evident in the resulting program, a **MEMORY PERFORMANCE ASSESSMENT** the developers uses post-hoc analysis to identify memory use hot spots and remove them.

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.

Components that use **FIXED SIZE MEMORY** are much easier to budget than those using **VARIABLE SIZE MEMORY**.

Systems that satisfy their RAM or secondary storage memory budget when they're started may still gradually 'leak' memory over time, so you'll need to Plug the Leaks as well.

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

Featurectomy Pattern

Also known as: Negotiating Functionality

How do you ensure you have realistic requirements for a constrained system?

- Software is 'soft', so the costs of extra functionality are hidden from those who request it.
- Extra functionality costs code and often extra RAM memory
- Specification teams and clients have a vested interest in maximising the functionality received.
- Some functionality confers no benefit to the users.

Software is soft; it's infinitely malleably. Given sufficient time and effort you can make it do virtually anything. But it costs time, effort and memory to achieve this.

Non-programmers are often unaware of this cost (programmers too!). And even if they are aware, or are made aware, many have no means of knowing exactly what the costs are. Will it take more memory to speed up the network performance by 50% than to add a new handwriting input mechanism? It's difficult for a non-programmer to know.

And in most environments the development team - and particularly the specification team if there is one - is under very great pressure to add as much functionality as possible. Functionality, and elegant presentation of functionality, is the main thing that sells systems. From the point of view of the client or specification team the trade-off is simple: if they ask too little functionality they may be blamed for it; if they ask for too much, the development team will take the blame if they don't deliver it. So the pressure is on them to over-specify.

Yet it's rare that all the possible functionality specified is essential, or even beneficial. For example some MS Windows applications contain complicated gang screens; MS Word 6 even includes an entire undocumented adventure game, hidden from all but initiates. Many delivered systems retain some of their debugging code, or checks for errant – and impossible – behaviour. Such additional code costs both code and often RAM memory in the final system. Yet they provide no service at all to the user.

Therefore: *Negotiate a specification to satisfy users within the memory constraints*

Analyse the functionality required of the system both in terms of its priority (how important is it?) and in terms of its memory cost (how much memory will it use?). Based on that, negotiate with your clients to remove – or reduce or modify – the less important and more memory intensive features.

Ensure that you remove any additional code for testing and debugging when you make a final release of the software.

Consequences

The released software needs to do less, so uses less ROM and RAM memory. In systems that implement Paging, the smaller code and memory sizes make for less disk swapping, improving the system's time performance.

There is less functionality to develop and to test, giving reduced development and testing. Because there is less functionality, there can be less interaction between features, leading to a more reliable, and often more usable, system.

However: The system has less functionality, potentially reducing its usability.

Unless the negotiation is handled carefully, the development team can be seen as obstructive to the client's goals, reducing client goodwill.

Implementation Notes

It can be difficult to impress on even technically-minded clients that memory limits are a significant issue for a project. Most people are familiar with the functionality of a standard MS-Windows PC, and find it difficult to appreciate the impact of much lower specification systems.

A good way to approach the negotiations is to prepare a Memory Budget allocating memory costs to each item of functionality – see *Functionality a la Carte* [Adams95]. Although this can of course be difficult to do, it makes negotiation straightforward.

Given this shopping list, and the fixed total budget, then the specification team and customers can make their own decisions about what functionality to include. Often they will have a much better idea of the importance of each feature, so they can make the trade-offs between options.

The Next Release

Frequently people (clients or developers) become 'wedded' to features, perhaps because it was their idea, or because somebody powerful wants it. In that case it becomes very difficult to negotiate such features out of a product no matter how sensible it may appear to everyone else concerned.

In that case a common approach is to agree to defer the feature until the next system release. By then it may well be obvious whether the feature is necessary, but also it will allow a more impartial appraisal once time has gone by.

Supporting Variant Systems

Features that are essential to one set of users may be irrelevant to others. In many cases there won't be any single user who needs all the system functionality.

So you can provide optional features in separate **PACKAGES**, which can be left uninstalled or merely not loaded at run-time. In systems that don't support packages, you might use conditional compilation or source code control branches to tailor different systems to the needs of different sets of users.

Sometimes this results in two-tier marketing of the system: a base-level product with low memory demands, and a high-tier ('professional') product with higher hardware requirements.

Thin Client

One particularly powerful form of **FEATURECTOMY** is possible when there is some form of distribution with a central server and one or more client systems. In such 'client-server' systems the trend until recently has been to have much of the business processing at the clients ('fat clients'), talking to a distributed database. This approach obviously requires a lot of code and data in the client. And it may well be unsuitable if the client has little memory or processing power.

Instead, given such a system, it is often possible to offload much of the processing to the server. You can do this by making the client simply be a graphics workstation (provide an character or X-windows terminal emulation). But often a better approach is to implement a 'thin client', which provides a UI and does simple user input validation, but which passes all the business-specific processing to a central server.

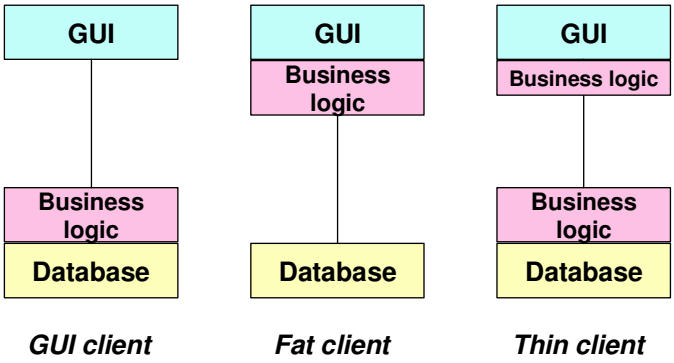

Figure 2: Three kinds of client-server

Other Featurectomy options:

Often there are specification alternatives to simply cutting out a feature altogether. For example you might agree **LOWER QUALITY MULTIMEDIA** for the implementation, or use **FIXED USER MEMORY** in the interface, to reduce the memory demands. You might **MAKE THE USER WORRY** – for example by making the user to explicitly start any functionality required, rather than starting it automatically.

You may be able to cut down on the additional data demands of the system. For example a mapping application might store only a subset at any time of all the maps required; a dictionary application might support only one language or technical subset at a time; an operating system might cut down on the number of simultaneous services available.

Debugging Code

One key set of users whose needs are different from others is the programmers themselves testing and debugging the system. Examples of such code are:

- Tracing code, to show what the program is doing.
- Checking code, to verify that the program is working correctly. Examples are assertions and invariants [Meyer]
- Debugging test harnesses, such as 'main()' functions added to classes for localised testing.
- Debugging support functions, such as functions to allow test code to access 'private' data for 'white box testing' [test]
- Instrumentation macros for memory and performance optimisation (see the Plugging the Leaks Pattern).

Clearly none of this code is vital to the delivered system. It will waste code space, and potentially impact the time performance of the system. So you'll want to remove it from the deliverable product. Ideally, though, you'll want to keep it in your codebase, so that it's available for future testing, debugging and optimisation.

The Eiffel language [Eiffel] is designed specifically to support this kind of dual mode. In debugging, it encourages programmers to define additional checking code: preconditions and postconditions for each function, and invariants for each class. In release mode the compiler doesn't generate this checking code at all.

Conditional Compilation in C++

In C++ the usual technique is to use pre-processor flags and macros. For example

```
#ifdef DO_TRACE
#   define TRACE( x ) printf( x )
#else
#   define TRACE( x )
#endif
```

allows us to use the TRACE throughout the code. When debugging, we can declare the DO_TRACE macro (in a global header file, or on the compiler command line); in the final system we can omit it.

An even more common form of this in C++ is the assert macro, built into the C++ environment (header file assert.h):

```
#ifdef NDEBUG
#   define assert(exp)    ((void)0)
#else
#   define assert(exp) (void)( (exp) || (_assert(#exp, __FILE__, __LINE__),
0) )
#endif  /* NDEBUG */
```

The _assert() function here displays a text message with the text of the assertion, and the location (sometimes in a dialog box). Then you can use expressions like:

```
assert( x== 0 );
```

and in debug mode the assertion is tested; in release mode the line of code is **FEATURECTED**.

Conditional Compilation in Java

Conditional compilation in Java uses a compiler optimisation. Most Java compilers can detect when certain code is 'dead' and will not produce corresponding byte codes. So a test using a static final boolean provides conditional compilation:

```
class Assertions  {
    public static final boolean isEnabled = true;
    // Change to false for release


    public static void assert( boolean assertion, String message ) {
        if (!assertion)
            throw new Error( "Assertion failed: " + message );
    }
}
```

You might use this as follows:

```
    public static void main( String[] args ) {
        try {
            int x=0;
            if (Assertions.isEnabled)
                Assertions.assert( x == 1, "x is non-zero" );
            Assertions.assert( x==1, "second one" );
        } catch (Throwable e) {
            e.printStackTrace();
        }
    }
```

Examples

[Matrix for Strap-it-on showing features, estimated peak and average ROM and ROM use to support each, and development time in man-days]. Based on this we decided to exorcise feature X.

Known Uses

In a recent Symbian EPOC mobile phone development, the initial ROM demands were way over budget. The development team used a **ROM BUDGET** and **MEMORY**

TRACKING to analyse the problem, and negotiated **FEATURECTOMY** with the client's specification team. In particular they agreed to have different language variants of the system for different markets, thereby considerably cutting down total size of **RESOURCE FILES**.

Microsoft Windows CE provides pocket versions of MS Word, MS Excel and other applications. In each application, the CE developers have cut down considerably on the functionality. For example Pocket Word 2.0 omits, amongst many other things, the following features of MS Word 97:

- Thesaurus
- Mail-Merge
- Auto Format
- Clip-art
- Support for non-TrueType fonts
- Float-over-text pictures

Pocket Word also uses a different internal file format from any MS Windows version of Word, **MAKING THE USER WORRY** about file conversion.

See Also

Usually you will need a **MEMORY BUDGET** as a basis for Featurectomy negotiations.

MEMORY TRACKING allows you to see the effects on memory use as features are implemented. Featurectomy may be appropriate if things look bad.

Some forms of **UI PATTERNS** may provide Featurectomy without significantly affecting the usability of the system. For example **FIXED USER MEMORY** provides feature with a fixed maximum memory use. And **USER MEMORY CONFIGURATION** allows the user to chose which features are present at run-time.

Alternatives to Featurectomy include **COMPRESSION**, using **SECONDARY STORAGE**, **PACKED DATA** and **SHARING** – and indeed most of the other patterns in this book.

Memory Tracking Pattern

A.k.a. Memory Accountant, Continuous Programmer Feedback

How do you find out if the implementation you're working on will satisfy your memory requirements?

- You're doing a project that may fail if the memory requirements exceed certain limits.
- The development team needs continued motivation to restrict memory use.
- Many teams are hostile to the formality of a full Memory Budget.
- If things are going well there's no point in going to unnecessary effort.

You are working on a software development project for a system with limited memory. Bad things will happen if the final system exceeds its memory constraints.

Yet the programming team – including yourself – may find it difficult to judge how important the problem is. People who've only worked in relatively unconstrained environments often have difficulty coming to terms with tight memory limits and the different programming styles these imply. Alternatively, they may overestimate the danger, and waste effort on unnecessary memory optimisation.

If you're working in a relatively informal environment, you may find that a detailed Memory Budget – with its culture of advanced planning and estimation – may not be welcome to your co-developers. They may resist the process, or simply ignore the results. Yet you still need to bring home the need to Think Small and to design the system to satisfy the memory constraints. Even if the team are willing participants in a budgeting process, that can involve lots of effort and overhead to construct and maintain budgets —more so if you are in a formal environment with lots of paperwork.

Alternatively, if you wait until almost the final system release and do a Memory Performance Assessment, then there's a possibility your designs and implementation may be too inflexible to allow much improvement at the last moment. What should you do?

Therefore: *Track the memory use of each release of the system, and ensure that every developer is aware of the current score.*

With each significant release of the system, use tools to examine the memory use of the entire system and – as far as possible – of each component within it. Publish this knowledge to all of the team.

Use graphs to show how the memory use varies between releases – ensure that everyone understands that a downward pointing graph is desirable, and label any major changes in the memory use of a component with a brief explanation.

If necessary, track the various worst-case scenarios (see Memory Budget), and track separately the figures for each one. If you have a Memory Budget, then compare the current figures with the targets in the budget. Consider creating a memory accountant role to perform this memory tracking. There are advantages if this role is not filled by the main project manager or technical lead — partly to lower their workload, but also to reduce the impression of management checking up on programmers.

Consequences

The feedback of their current memory status encourages every programmer to **THINK SMALL**, without a need to impose the formal limits of a **MEMORY BUDGET**. Doing just **MEMORY TRACKING** can also take *less programmer effort* on an ongoing basis than full budgeting, since programmers will deduce the need to save memory for themselves. So this pattern can be very effective in less formal development environments, creating self-imposed *programmer discipline*, which will help reduce the *absolute memory requirements* of the system.

Having figures for memory use increases the *predictability* of the memory use of the resulting system. It highlights potential *local* problem areas early, so you can address them or schedule time for a Memory Performance Assessment.

If you can produce figures for the separate system components, then you can establish the actual contribution of each component, showing the *local* contribution of each to the *global* memory use.

However: Tracking memory use and producing the feedback needs *programmer effort*. Measuring the memory use may require *hardware or operating system support,* or mean further *programmer effort* to instrument the code accordingly, especially to measure RAM allocation and use.

There's a danger that early figures, when the functionality is incomplete, may be misleadingly low. Or that you may have chosen an unrepresentative set of worst-case scenarios. Either of these factors can cause the figures to be over-optimistic, lulling the team into a false sense of security and discouraging future *programmer discipline*.

Implementation Notes

How do you do the measurements? This section examines tools and techniques to find out the memory use of the various components in a system.

As discussed in the Memory Budget pattern, there are several types of memory use you may want to track including: Total RAM memory, 'Live' RAM usage, ROM use and Secondary storage.

External Tools

It's usually straightforward to measure ROM use – just examine the sizes of the files that make it up, and the size of each ROM image itself. However how are you to measure the memory use that varies with time?

Similarly in most environments you can measure the secondary storage used by the application at a given time using the file system utilities.

Many environments also provide utilities to track the RAM use of each process. For example, Microsoft Developer Studio's Spy++, [MicrosoftSpy97], allows you to examine the memory use of any process under Windows NT. Figure 3 shows an example display. The important figures are "Private bytes", which gives the current heap memory use for the process, and 'peak working set' (see the Paging pattern) which gives the minimum RAM that might be needed.

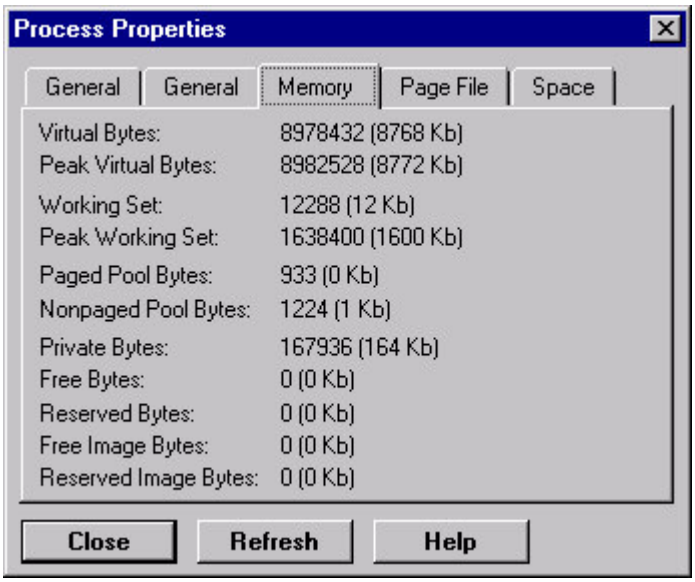

Figure 3: Microsoft Spy++ Display for a Process

Unix systems similarly provide tools such as `ps` and `top` to list processes' memory use.

The EPOC operating system supports rather more memory-constrained systems, so provides a rather more detailed display of memory use using its 'Spy' tool:

Thread name	TID	Pri	HS	HU	SS	SU	AS
EFile[100000bb]::Main	4	10	64k	0	8k	N/A	
LoaderThread	5	0	8k	0	16k	N/A	
Main	8	0	120k	0	8k	N/A	
FbServ[100001ee]::Main	10	0	64k	0	8k	N/A	
System	12	0	56k	25424	12k	5240	
Brdsrv[00000000]0001::Main	14	0	8k	2024	4k	1068	
System::SystemServerThread	15	400	28k	17072	8k	4868	
Main	8	0	120k	0	8k	N/A	
Ealwls[100001db]0001::AlarmWorldServerThread	19	0	12k	8192	8k	1824	
AppArcServerThread	20	400	20k	14916	8k	4096	

Figure 4: EPOC Spy Display

The EPOC Spy Display in Figure 4 shows for each thread, the total heap size (HS), the RAM allocated within each heap (HU), the Stack size (SS), and – where the process protection permits – the maximum stack usage (SU).

Code Instrumentation for RAM use.

System tools can normally only show you an external view of each separate process, showing its total memory use. Perhaps your important components are more fine-grained than individual processes, or perhaps there are no system tools available for your particular environment. What should you do then?

The solution is to 'instrument' your own code, adding test code to track memory allocations and de-allocation. See the Memory Limit pattern for details.

Another possibility, if your environment supports it, is to use separate Heap structures for each component, and use the system tools to examine the memory use

of each heap. This is possible using EPOC's SPY tool – for example Figure 5 shows three heaps owned by the Font Bitmap Server process (FONTBITMAPSERVEER:$STK, FBSSHAREDCHUNK and FBSLARGECHUNK)

Heap name	Heap start	Size
EPOC[00000000]0001::SUPERVISOR	07C0000	260K
KERNELWINDOW::$STK	09C2000	12K
MAIN::$STK	0FF4000	132K
FILESERVER::$STK	1902000	68K
LOADERTHREAD::$STK	1A54000	8K
WSERV::$STK	2082000	500K
FONTBITMAPSERVER::$STK	2592000	68K
FBSSHAREDCHUNK	27A0000	332K
FBSLARGECHUNK	67A0000	120K
SYSTEM::$STK	A8E2000	68K

Figure 5: EPOC Spy: Heap Display

Code Instrumentation to find Maximum Stack use

There are at least two approaches to estimating the maximum stack use of a thread.

You can use compiler tools instrument each function call explicitly. For example in debug mode Microsoft Visual C++ inserts calls to a function _chkstk() at the start of each function. If you can replace the (undocumented) implementation of this function, then it's straightforward to find out the current stack use. Many other compilers support similar features.

A less intrusive approach is to ensure the unused portion of the stack is filled with a random value, possibly zero. At the end of the process – or at any other time – you can track to see how much of this memory has been overwritten. EPOC, for example, initialises all stacks to an arbitrary 29 hex, and EPOC Spy uses this when it examines the stack of each readable process to discover its used stack (SU in Figure 4).

Case Study

[Example of the same system as Memory Budget, showing graphs of actual measured memory use plus target for three releases, for several components, both worst case and normal. How do we combine the graphs? this sounds great!]

Known Uses

The EPOC operating system aims to support systems with constrained hardware. In particular, some of the target platforms have hard limits on their ROM memory for storing code. To explore this problem, the design team recently produced a Memory Tracking document, identifying all the components and showing their ROM use over successive releases of the operating system. The document discusses the reasons for each component size change – typically, of course, increases in functionality. It provides an excellent basis for a Memory Budget for any future system, both by suggesting the memory use of combinations of components, and by suggesting the possible gains from featurectomies in specific components.

Brooks?

Roxette project, of ROM.

See Also

Memory Budget can provide a set of targets for developers to compare with the tracked values. Memory Performance Assessment uses similar techniques to determine the memory use at a particular point, as a basis for memory optimisation.

The Memory Limit pattern describes techniques to track memory allocation within your programs at runtime.

Extreme Programming (XP) advocates continual tracking of all aspects of a development project, and stresses the advantages of the auditor role not being part of the project's power structure [Beck, 1999].

Memory Optimisation Pattern

A.k.a. Memory Performance Assessment.

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

- You're developing a system that may turn out to be memory-constrained, or you're porting an existing system to a more constrained environment.
- You don't want to devote too much effort to reducing memory requirements early in the project – it may prove unnecessary.
- Often other constraints may be more important than memory constraints
- When the system is near to release, memory performance does turn out to be a problem.

Your system has to meet particular *memory requirements*, but other requirements are more important. A full-scale Memory Budget or Memory Tracking would cost *programmer discipline* and *programmer effort* that could 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 controlling the amount of memory the system occupies.

Yet there's a reasonable likelihood that the system's memory constraints may prove a problem. How do you allow for this possibility in the development?

Therefore: *Implement the system normally, then optimise memory use afterwards.*

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 usual, with not much special attention paid to memory use. You'll usually do a quick informal **MEMORY BUDGET** very early on, just to make sure there are no really pressing concerns. And where the implementation and design costs are reasonably low, you'll use patterns to reduce the memory use – but only where these don't conflict with more vital design constraints.

If the resulting system meets its memory requirements, then that's as far as you need take matters. But if, as often happens, the program does not meet its *memory requirements* you to do a Memory Performance Assessment.

By an examining the code, using profiling tools, and any other appropriate techniques, find out where the system is using most memory. Identify the most promising areas to improve memory use, and implement changes accordingly. Repeat this process until the system satisfies its memory constraints.

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*.

Initial development can be less constrained, improving programmer *motivation.* In addition the performance assessment is a single, short-term task, for the team to achieve – also improving programmer *motivation*.

However: The *memory requirements* of the resulting program will be hard to *predict.* In many cases it requires more *programmer effort* to leave memory optimisation to last than performance optimisation would, because memory optimisation tends to require changes to object relationships that can affect large amounts of code.

Memory optimisation after implementation is more likely to compromise the design, leading to poorer *design quality* than in systems designed from the start to support memory restrictions. Local optimisations may also compromise the global integrity and overall architecture of the system.

Finally the optimised system will need testing, increasing the total *testing cost.*

Implementation Notes

Static much easier than dynamic

Static Analysis

Code analysis. What structures are there? How much memory will each use?

Code and static data size – look for redundant code (execution trace tools; human examination; linker map tables).

Look particularly for any object that will have very large numbers of instances.

Memory Profiling Techniques

Memory Tracking discussed techniques to find the total memory used by each component. How do you examine each component to find out whether and how it's wasting memory?

Tracing as objects are created and destroyed. CodeXXXX shows objects being created and destroyed.

Heapwalk tools (Windows, EPOC) show the objects.

Optimisation Approach

Low hanging fruit.

you should only optimise as and when needed, never in advance. you should define measurable performace criteria, optimise until you meet them, and then stop --- backing out other optimisations if they don't contribute much to the final result.

* you should be able to undo optimisations, and should undo them when they are no longer needed.

* cite the Lazy Optimisation pattern (plopd2 or 3, it's in the almanac) LOTS. if you have time, its worth a read to get their mindset, which seems right on the button.

Optimisation Patterns

Groups of objects – all.

Local changes: Packed data, Memory Pooling, Multiple Representations.

Compression: String Compression, File Compression, (Tokens)

Secondary Storage: Packages, Data Chaining (local temp file)

What is and is not safe to leave until Memory Optimisation

Leave only local optimisations.

Small Interfaces don't leave. Partial Failure. Fixed DS.

UI Don't leave (except Notifier)

The deal here is interfaces vs. implementation. You can optimise implementations locally. You have to optimise interfaces globally, which often translates to --- you can't optimise them. also, the flip side of this is, what optimisation techniques are (and are not) safe to do? I.e. you don't want to break modularity when you optimise a local implementations, or else you can't undo them.

Example

[Regclean] provides an example of an assessment of the redundant memory use in an non-optimised workstation product. Quote it?

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. Another common situation is in porting an existing system to an environment with insufficient memory to support the system unchanged. A memory performance assessment must be part of the porting process.

For example in the development of the Rolfe&Nolan 'Lighthouse' system to capture financial deals, the team developed the terminal software and got it running reliably. However they then discovered that some users required to keep information about huge numbers of deals in memory at one time – and the cost of paging the memory required made the performance unacceptable. The team did an informal memory assessment, and optimisation. They found they had lots of small, heap-allocated objects. So they purchased an improved heap management library with a much smaller overhead both for each allocated block and for fragmentation. They identified that they didn't need the full information for each deal, so they used Multiple Representations to reduce the memory each deal takes by default.

[Blank+95] describes the process of taking an existing program and optimising it to run under more constrained memory conditions.

See also

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. If the system is going to undergo continual change, then you should do Memory Tracking to ensure that members of the programming team are continually aware of the memory requirements.

Plugging the Leaks Pattern

Also known as: Alloc heaven testing.

How do you ensure your program recycles memory efficiently?

- Programmers aren't perfect.
- Heap-based systems can make it easy to have memory leaks
- Systems will run happily and satisfy users even with memory leaks
- But leaks will still progressively drain free memory from the system.

Programmers aren't perfect. It's easy to make mistakes [whyMistakes], and if there's nothing that points out the mistake, very difficult to spot and correct them. An important type of such 'stealth errors' in O-O systems is called a 'memory leak': a heap-based object or data structure that's neither still required by the program, nor returned to the heap.

In C++ and languages without garbage collection this situation is very easy to achieve; allocated blocks only return to the heap if you use the memory freeing mechanism (delete in C++). If the program discard all pointers to a heap object, the object becomes 'lost' – the program will never be able to delete it and return it to the heap. In the phrase used by EPOC developers, the object goes to 'Alloc Heaven'.

[Picture showing continuous heap, with pointers from 'program' and internal, and an orphan block]

Even Java and languages with garbage collection can have memory leaks. The Garbage Collector will automatically clean up objects discarded to Alloc Heaven as above; however it's quite common to have collections still containing pointers to objects that are no longer actually required. GC can't delete these objects, so they remain – they are memory leaks.

Now programs will run happily and satisfy users even with memory leaks. Most Microsoft software, for example, appears to have minor memory leaks under some circumstances. But leaks use up valuable memory that may be needed elsewhere; leaks that are significant over the time of a given process will impact performance and push the system over it's Memory Budget.

Therefore: Test your system for memory leakage and fix the leaks.

During program testing, use tools and libraries to ensure that you detect when objects left on the heap that are no longer required. Track these 'leaks' down to their source and fix them.

Also test to ensure that your secondary storage doesn't progressively increase – unless this is a necessary feature of the system – and fix the problem if it does.

Consequences

The application uses less memory. That makes it less prone to memory exhaustion, so it is more reliable and more predictable. More memory is available for other parts of the system. Applications running in Paged systems will have better time performance.

Fixing memory leaks often solves other problems. For example a 'leaked' object may own other non-memory resources such as a file handle, or do processing when it's cleaned up. Fixing the leak will also solve these problems.

However fixing the leaks requires programmer effort.

The task of fixing the leaks provides no obvious benefit to customers of the system (since all the benefits are indirect) who may resent the time 'wasted'.

Implementation Notes

Determining whether there are Memory Leaks

The most common way to find out if there are memory leaks is to do stress testing [BeizerXXX], and use Memory Tracking tools to see if the heap memory use gradually increases. However, unless the stress testing is very rigorous, this won't find memory leaks caused by very exceptional situations (such as memory exhaustion).

Note, however, that in some environments (C++) heap fragmentation will also cause the heap memory use to increase. If memory use is increasing, but the techniques we describe below don't find any 'leaked' objects, then the problem may well be fragmentation; you can reduce it using Memory Compaction or Fixed Data Structures.

Causes of C++ Memory Leaks

The most common cause of C++ memory leaks is 'alloc heaven': the program discards all references to an object or block of memory without explicitly invoking the destructor.

The best way to fix such leaks is to identify the objects involved, and to insert the appropriate code in the correct place to delete the object. This correct place is often defined in terms of 'ownership' of the object [Ownership].

If there are many small memory leaks, you may well choose to abandon trying to fix them all piecemeal and use Memory Discard (using a scratch heap and throwing the entire heap away) instead.

Causes of Java Memory Leaks

The normal cause of memory leaks in Java is the program retaining spurious references to objects that are in fact no longer required. Ed Lycklama [Lycklama99] calls such objects 'Loiterers', and identifies and names the four most common reasons:

- Lapsed Listener – object added to collection, never removed.
- Lingerer – reference used transiently by long term object.
- Laggard – object changes state; references refer to previous state objects.
- Limbo – stack reference in a long running thread.

The normal way to fix the last three of these is either to rearrange the code so that the situation doesn't happen, or simply to set the offending reference to null at the appropriate point in the code.

Finding Memory Leaks using Checkpointing

The most common way of tracking down memory leaks is to use a memory checkpointing technique. This technique relies on you being able to define two 'checkpoints' in the code, such that all memory allocated after the first checkpoint should be freed before the second checkpoint. You then insert 'probes' (usually function calls or macros) at both checkpoints to verify that this is so.

There are two possible techniques to implement checkpointing. The most common approach is for the heap implementation to support it directly (usually only in debugging mode). Then the first probe stores the state of the heap at that point – typically creating an array of references to all the allocated blocks. And the second probe verifies that the state is unchanged – typically by checking that the set of allocated blocks is the same as before. Of course in garbage collecting languages like Java, both checkpoint functions must invoke a garbage collection before doing their heap check.

The alternative approach, if the heap implementation doesn't support checkpointing directly, is to keep a separate collection of pointers, and ensure that every new object created is added to this collection, and every object deleted is removed from it. Then the first checkpoint function creates this collection, and the second tests if it's empty.

The problem with this is how to intercept every allocation and deletion. In C++ you can implement this by implementing debugging versions of the new and delete operators. In Java you can use a weak reference collection, and replace the constructor for Object to add each object to this collection.

The simplest and most common checkpoints to chose are the start and end of the application or process. Many environments do this by default in debug mode; EPOC applications compiled in debug mode will display a fatal error ('Panic') dialog box with the memory address of objects not deallocated; Microsoft's MFC environment displays information about all remaining allocated objects to the debugging window.

Tracing the Causes of Memory Leaks in C++

The checkpointing technique above will give you the memory references of the leaked objects, but it doesn't tell you why the memory leak occurred. To find out more, you need to track down more information about the objects.

In C++, a good debugger can usually track down the class of an object with a vtbl if you know the likely base class. For example, in EPOC most heap objects derive from the class `CBase`; if the leaked object has address `0xABABABAB`, then displaying `(CBase*)0XABABABAB` will usually show you the class of the allocated object. In MFC, many objects derive from `CObject`, so you can use the same technique.

More helpful, and still simple, MFC defines a macro:

```
#define DEBUG_NEW  new( __FILE__, __LINE__ )
```

The pre-processor expands `__FILE__` and `__LINE__` macros to the file name and line number being compiled, and the corresponding debug versions of the `new` operators – `new( size_t, char*, int)` store this string and integer with each memory allocation. So if you put the following in a header file:

```
#define new DEBUG_NEW
```

Then the tracing information in the checkpointing heap dump can include all the memory items allocated.

Best of all, if possible, is to use a specialised memory tracking tool, such as Purify [Rational] or BoundsChecker [NuMega]. These tools implement their own versions of the C++ heap libraries, and use debugging techniques to track memory allocation and deletion – as well as tracking invalid references to memory. They also provide programmer-friendly displays to help you track down specific memory leaks.

Tracing the Cause of Memory Leaks in Java

Having located a leaked object in Java, you need to track back to see what objects, static collections, or stack frames still have references to it.

The best way to do this is to use one of the several excellent Java memory checking tools: JProbe from KL Group [jprobe], [Apicella99]. OptimizeIt from Intuitive Systems [optimizeit], or Heap Analysis Tool from Sun [hat]. No doubt others will be available shortly.

Both JProbe and OptimizeIt work by modifying the Java kernel to provide better debug heap management, and provide very user-friendly GUIs to help debugging – see Figure 6, for example; HAT uses a detailed knowledge of Sun's own heap implementation.

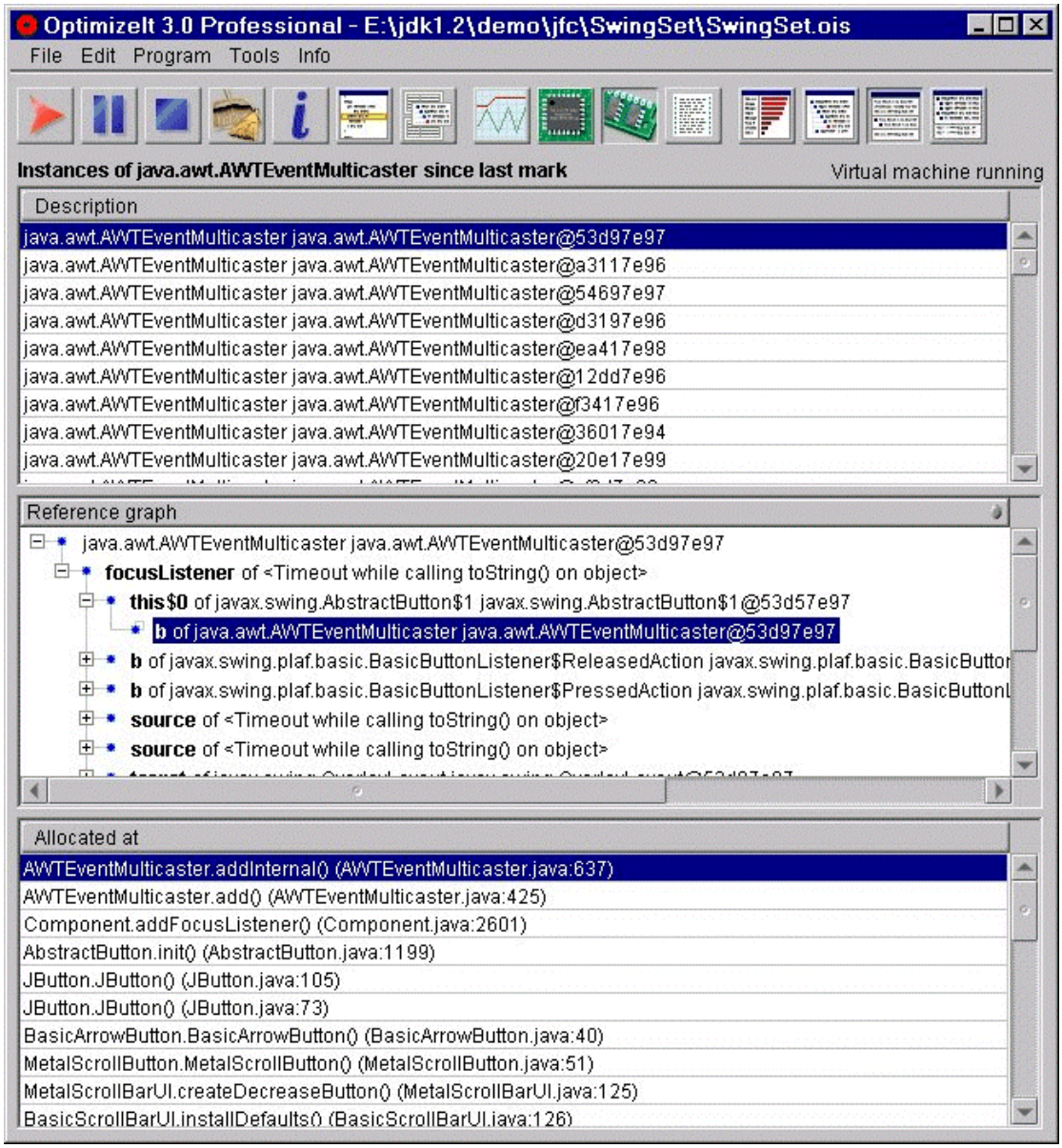

Figure 6: Tracking references to a Java object using OptimizeIt

Examples

EPOC Exhaustion Test Extension

In most C++ applications, the most likely cause of a memory leak is an exception that destroys stack pointers to heap memory. As discussed in the Exhaustion Test pattern,

such memory leaks are difficult to simulate by normal testing but easy with Exhaustion Testing.

This example extends the example code we quoted in the Exhaustion Test pattern to check for memory leaks during each test. It uses the EPOC pair of macros __UHEAP_MARK and __UHEAP_MARKEND. In debug mode, these check that the number of cells allocated at MARKEND is the same as at the corresponding MARK, and display a message to the debug output if not. The pairs of macros can be nested. __UHEAP_MARKEND also checks the heap for consistency.

```
static void MemTest(void (*aF)())
   {
   __UHEAP_MARK;
   for (int iteration=0; ;++iteration)
        {
        __UHEAP_FAILNEXT( iteration );
        __UHEAP_MARK;
        TRAPD(error,(*aF)());  // Equivalent to try...catch...
        __UHEAP_MARKEND;
        if (error==KErrNone)   // Completed without error?
             {
             test.Printf(_L("\r\n"));
             break;
             }
        else
             {
             test.Printf(_L("  --  Failed on %d\r\n"),iteration);
             }
        }
   __UHEAP_MARKEND;
   __UHEAP_RESET;
   }
```

MFC Example

The following pieces of code implement checkpointing using the Microsoft MFC class, CMemoryState. The first checkpoint (corresponding to EPOC's __UHEAP_MARK above) stores the current state:

```
#ifndef NDEBUG
CMemoryState oldState;
oldState.Checkpoint();
#endif
```

The second checkpoint (corresponding to EPOC's __UHEAP_CHECK) checks for differences and dumps details of all the changed objects if it finds any:

```
#ifndef NDEBUG
CMemoryState newState, diffState;
newState.Checkpoint();
if (diffState.Difference( oldState, newState ))
{
   TRACE( "Memory loss - file %s, line %d\n", file, line );
   diffState.DumpStatistics();
   diffState.DumpAllObjectsSince();
}
#endif
```

Known Uses

Plugging the Leaks is a standard part of virtually every project with memory constraints.

See Also

Exhaustion Testing often shows up leaks to plug.

If there are many difficult-to-find small leaks that don't really impact the system in the short term, it can be much easier to use Program Chaining – terminating and restarting a process regularly – instead of Plugging the Leaks.

Exhaustion Test Pattern

Also known as: Out-of-memory testing.

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

- Functionality that isn't tested probably won't work correctly.
- The functionality of a system includes what it does when it runs out of memory.
- It's difficult to make a program run out of memory on a normal system.
- It can be difficult to reproduce errors caused by limited memory.

Programs that deal gracefully with resource failures — say by using the **PARTIAL FAILURE** or Fixed Memory patterns — have a large number of extra situations to deal with, because each resource failure is a different execution event.

Ideally, you'll test the program in every situation that will arise in execution. But 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.

You might create out-of-memory situations by allocating a lot of memory so that the system is resource-strapped and the errors do happen. This has two problems. First it's difficult to get exactly the same situation each time. So it will be difficult to reproduce any errors you have. Secondly, in many environments you'll be running your debugging and possibly development systems in the same environment (and maybe processes belonging to other users of the system too). Forcing the system to be short of memory will prevent these tools from working correctly as well.

Therefore: *Use testing techniques that 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 all 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 can 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.

It will be easier to replicate the errors, making it easy to debug the problems and verify any fixes. Which reduces the total programmer effort required to get the system working.

However you'll still need a significant testing cost to be reasonably certain that the resulting program will work correctly. 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.

Implementation Notes

Simulating memory failure

Some systems, such as EPOC and the Purify tool for UNIX and Windows environments, provide a version of the memory allocator that can be set to fail either after a specified number of allocations or randomly. For other systems (such as MFC), it's easy to implement this in C++ by redefining the `new` operator.

Here's an example for the MFC environment. Note that it uses the `DEBUG_NEW` macro, which provides debugging information about allocated blocks (see Plugging the Leaks).

```
#ifdef MEMORY_FAILURE_TEST
BOOL MemoryFailureTime();
# define new MemoryFailureTime() ? (AfxThrowMemoryException(),0) : DEBUG_NEW
#else
# define new DEBUG_NEW
#endif

#ifdef MEMORY_FAILURE_TEST
static int allocationsBeforeFailure;
BOOL MemoryFailureTime()
{
    return allocationsBeforeFailure-- == 0;
}
#endif

BOOL TestApp::InitInstance()
{
#ifdef MEMORY_FAILURE_TEST
   allocationsBeforeFailure = atoi( m_lpCmdLine );
#endif
// ... etc.
```

An alternative is to implement a debug version of the function ::operator new(size_t) with similar behaviour.

Simulating memory failure by external actions

There are two approaches to setting up the test. One common approach is to provide a mechanism where user input to the application or system to cause it to fail. In the example above, the user passes an integer as the command line; allocation will fail after that number of memory allocations.

As an alternative the application might put up a dialog ("Will give allocation failure after how many allocations?") where the user enters a number. Then heap allocation fails after that number of allocations. In EPOC and Purify, this facility is built into the runtime debugging environment. In EPOC, for example, the keystroke XXX brings up this dialog.

The advantage of this approach is that it makes it easy to debug a particular situation without needing to perform all the other tests. It also works well with almost the entire system running normally. The disadvantage however is that for complete testing, the user will have to run the system a very large number of times, entering a different value each time. While it might be possible to automate this process by using external tools to simulate user input, this would be bound to be cumbersome and slow.

Repeatedly simulating memory failure using test harnesses

The alternative approach is to write a test harness that runs a particular function or piece of code repeatedly, with different values of the 'allocations until failure' value.

This approach is mandatory in a system with very strong test requirements, or one using the 'Extreme Programming' approach where every test remains as part of the development environment and all tests are run early and often.

This approach provides much more complete testing of the given functionality. But it's only realistic for specific functions, and ones that require no user or external input to execute.

Example

The following is a (much simplified) extract from the test code for part of an EPOC application – a class we'll call CApplicationEngine.

It uses the heap debugging facility, User::__DbgSetAllocFail(), to set up the application (`EUser`) heap for 'deterministic failure' – i.e. failure after a specific number of allocations.

The function MemTest simply calls a given function many times, so that memory fails after progressively more and more allocations. On EPOC memory failure, the code always does a 'Leave' (EPOC Exception – see Partial Failure), which the code can catch using the macro TRAPD. The test support RTest class simply provides output to the tester and to a test log.

```
RTest test;

static void MemTest(void (*aF)())
   {
   for (int iteration=0; ;++iteration)
        {
        __UHEAP_FAILNEXT( iteration );
        TRAPD(error,(*aF)());  // Equivalent to try...catch...
        if (error==KErrNone)   // Completed without error?
             {
             test.Printf(_L("\r\n"));
             break;
             }
        else
             {
             test.Printf(_L("  --  Failed on %d\r\n"),iteration);
             }
        }
   __UHEAP_RESET;
   }
```

The main function, DoTests(), calls MemTest for each testable function in turn:

```
static void DoTests()
   {
   test.Start(_L("Sheet engine construction"));
   MemTest(Test1);
   test.Next(_L("Test set and read on multiple sheets"));
   MemTest(Test2);
   // ... etc.
   test.End();
   }
```

And a typical testable function might be as follows:

```
static void Test1()
   {
   CApplicationEngine* theApplicationEngine = CApplicationEngine::NewLC();
   theApplicationEngine->SetRecalculationToBeDoneInBackgroundL(EFalse);
   CleanupStack::PopAndDestroy();  // theApplicationEngine
   }
```

Known Uses

Exhaustion testing is a standard part of the development of every component and application released by Symbian. The EPOC environment demands **PARTIAL FAILURE**, so each possible failure mode is a part of the application's functionality.

The Purify environment for C++ supports random and predictable failure of C++'s memory allocator. Other development environments provide tools or libraries to allow similar memory failure testing.

Symbian's EPOC provides debug versions of the allocator with similar features and these are used in module testing of all the EPOC applications.

See Also

Exhaustion Testing is particularly important where the system has specific processing to handle low memory conditions, such as the **PARTIAL FAILURE** and **CAPTAIN OATES** patterns.

C++ doesn't support garbage collection and normally throws an exception on allocation failure, so the most likely consequence of incorrect handling of allocation failure is a Memory Leak as the exception looses stack pointers to allocated memory. So C++ Exhaustion Testing is usually combined with Plugging the Leaks.

References

[Noble+Weir98] *Proceedings of the Memory Preservation Society - Patterns for Small* Machines James Noble. Charles Weir Proceedings of the conference EuroPLoP 1998

[Beizer84] *Software System Testing and Quality Assurance,* Boris Beizer, Van Nostrand Reinhold 1984, 0-442-21306-9

[EPOCSDK]

[whyMistakes] Something about the brain's stack of seven items. Tony Buzan? Use your head.

[Weir96] *Improve your Sense of Ownership*– Charles Weir, ROAD magazine, March 1996. http://www.cix.co.uk/~cweir/papers/owner.ps

[Apicella99] JProbe Suite 2.0 takes the kinks out of your Java programs, InfoWorld May 17, 1999 (Vol. 21, Issue 20) http://www.infoworld.com/archives/html/97-e04-20.78.htm

[OptimizeIt] http://www.optimizeit.com/

[Jprobe] www.klgroup.com

[sun] www.sun.com

[Lycklama99] *Memory Leaks in Java*, Ed Lycklama, Presentation at Java One 1999. Available at http://www.klgroup.com/jprobe/javaonepres/

[Regclean] A memory assessment of MS Regclean in the risks digest http://catless.ncl.ac.uk/Risks/20.37.html

[Brooks75] The Mythical Man Month.

[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

[MicrosoftSpy97] Microsoft Visual C++ 5.0 Spy++ Online Help Documentation, Microsoft 1997.

[Gilb88] Principles of Software Engineering, Tom Gilb, Addison Wesley 1988, 0-201-19246-2

[Flanders&SwanXX] The laws of thermodynamics (song). In 'the collected works of F&S',

DeMarco Project Planning?

[Filipovitch96] Project Management: Introduction, A.J.Filipovitch 1996, http://krypton.mankato.msus.edu/~tony/courses/604/PERT.html

[Baker&Baker98] Complete Idiot's Guide to Project Management, by Sunny Baker, Kim Baker, MacMillan 1998; ISBN: 0028617452

[Kruekeberg&Silvers74] KRUECKEBERG, D.A. & A.L. SILVERS. 1974. "Program Scheduling," pp. 231-255 in Urban Planning Analysis: Methods and Models. NY: John Wiley & Sons, Inc.

[Brooks75] The Mythical Man Month.

[PalmBudget] ID 1136 Palm web.

[Shaw&Garlan] *Software Architecture - Perspectives on a Emerging Discipline*, Shaw and Garlan, Prentice Hall ISBN 0-13-182957-2

[XP] Extreme Programming Explained, Kent Beck, Addison-Wesley 2000, 201-64141-6

Software Requirements and Specifications by Michael Jackson

Peopleware Tom DeMarco

[Adams95] Functionality a la Carte, S.S.Adams, Patterns for Programming Design 1, 7-8.

[Rappel] *to add in text*

[Episodes] to add in text

