Process Patterns for Small Systems

© Charles Weir and James Noble 1997-2000.

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 have been published in the book *Small Memory Software,* Addison-Wesley Software Patterns Series in 2000.

This paper contains the following patterns:

- Thinking Small
- Memory Budget

The full chapter will include several additional patterns (and a proper bibliography!); the first pattern includes an overview of all the patterns that will be included in final chapter, including these additional patterns.

Related patterns have appeared in several conferences, including:

- High-level and Process Patterns from the Memory Protection Society. James Noble and Charles Weir. In Pattern Languages of Program Design 4. Neil Harrison, Brian Foote and Hans Rohnert, editors. Addison-Wesley, 1999.
- Patterns for Small Machines. James Noble and Charles Weir. Proceedings of the European Conference on Pattern Languages of Program Design, Irsee, Germany. Universitäts Verlag Konstanz. 1998
- Secondary Storage. Proceedings of the European Conference on Pattern Languages of Program Design, Irsee, Germany. Universitäts Verlag Konstanz. 1999

We'd like to thank all those who have reviewed, comment on, or shepherded these patterns. In particular many thanks are due to Linda Rising, our Europlop 2000 shepherd for her comments on this draft.

Further information about the Small Memory Software project can be found on the web at: http://www.smallmemory.com

Major Technique: Thinking Small

A.k.a. 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. Perhaps you've done some back-of-the-envelope calculations and they've indicated that memory requirements may be a problem; perhaps managing the program's memory consumption is an explicit part of the project's brief; or perhaps you've worked on a similar project before, and that project had problems managing memory use.

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 spends lots of effort optimising everything to work in very limited memory, they'll waste a lot of time and can produce a product that could have been much better. Worse still, the resulting system may fail because they have been optimising the wrong thing. You can minimise RAM memory use, for example, but your system will fail nonetheless if it needs twice the ROM space that the hardware supports. Optimising memory can reduce time-performance, usability, and even the overall capacity of the system.

When building any system you have to moderate the demands of different components in the system against each other. This 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. Design decisions about the trade-offs based on individual designers' foibles, on gut feel, or on who shouts loudest, will lead neither to consistent successful designs, nor to a harmonious development.

To make matters worse, while some decisions can be deferred until later in the project, some decisions are pervasive; by providing a framework for later decisions they will affect every component and every line of the system you are building. Strategic decisions about memory use can reflect in the design of the interfaces between components, in the trade-offs between main memory, ROM, and secondary storage, and in the support for fault tolerance and error handling.

Therefore: *Think Small! Make the memory constraints a priority 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 and worst case values with clients. Meanwhile also estimate very approximately the likely memory needs of the system you're developing, and identify the tensions between the two. Determine

the key design decisions that significantly impact memory use and that set the style for the rest of the project, and ensure these decisions happen early.

Based on this analysis, identify which forces are most vital. The answer may be a constraint on one of the forms of memory in the system, but other forces – time performance, reliability, usability – may prove as much or more important than memory use. Enshrine these priorities as strategy for everyone working on the project. Ensure that absolutely everyone working on the team understands which forces are most important in the project. Write the strategy in a document; make presentations; preach to the choir; print the T-shirts! Indoctrinate each new developer who joins the team afterwards with these strategic 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 more 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 the responsiveness of the user interface, to ensure quick answers in dangerous situations. So the system's components and interfaces are designed first and foremost to minimise memory use, and then to give reasonable user response time.

Consequences

Every member of the team will understand the priorities. Individual designers will be *motivated* 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, and increasing the *coordination* of the project as a whole.

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 memory strategy takes *time* and effort at an important stage of a project, and *discipline* to keep to it later.

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

There are a few basic principles that encourage thinking small, and underlie the design of software for smaller systems:

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 extend the objects 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 [Auer and Beck 1996]

An excellent way to promote **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.

There are a number of other issues to consider when thinking small:

1. Memory Requirements vs. Memory Predictability. There are two independent, yet closely related forces that bear upon building small systems. The most obvious is the absolute *memory requirements* of the system, that is, simply the actual amount of memory the system requires to run. But, while minimising a program's memory requirements can make it less likely to run out of memory, to be able to ship a system you need to be confident that it can run successfully in a given amount of memory. That is to say, you often need to be able to *predict* the system's actual memory consumption in advance.

Because many patterns (such as **FIXED ALLOCATION**) increase a program's memory consumption to gain predictability, whilst others (such as **VARIABLE ALLOCATION**) do the reverse, you need to consider the importance of absolute requirements against predictability when drawing up your project's memory priorities. How you do this will depend on the project you are working on. A hand-held video game, for example, can save its high score and restart if it runs our of memory; but the operating system for an embedded life-critical system may need to guarantee that it can operate for years without failing.

2. Implicit Strategies. Many projects work in a well-understood context, so it's not necessary to make the your project's strategy explicit. For example an MS-Windows shrink-wrapped application can assume a total system size of more than 16Mb RAM (and around 32Mb paged memory), and at least 50Mb disk space for the program – as we can see by studying any number of industry standard applications.

Windows developers share an unwritten understanding of the acceptable memory requirements for 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, and 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.

Using the same 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 physical memory than other applications.

3. Developers from Different Environments. Programmers and designers used to one environment often have very great difficulty changing to a different one. For example, MS Windows programmers coming to the EPOC or Palm operating systems often have great difficulty internalising the idea that programs must run indefinitely and cannot afford to stop when they run out of memory. Windows CE developers have even more of a problem with this, as the environment is superficially similar to normal Windows. If programmers do not adapt to the new environment, the resulting code will be poor quality and is unlikely to satisfy users' needs.

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.

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.

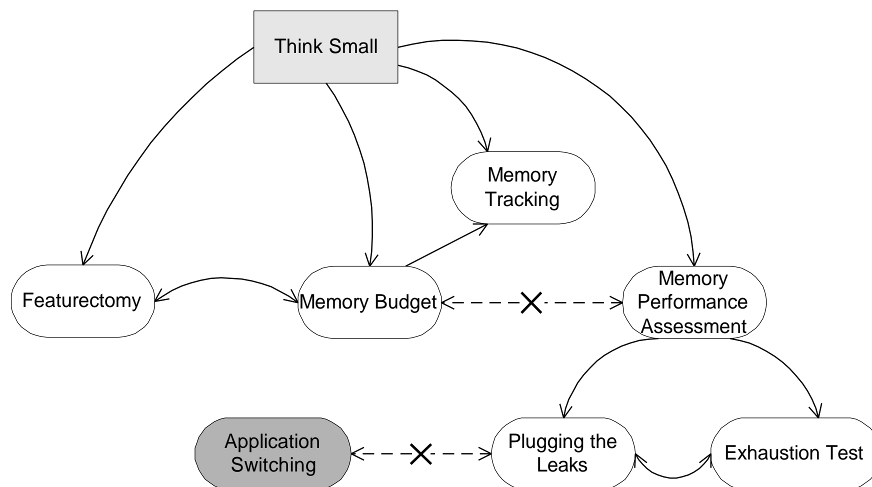

Figure 1: Process Pattern Relationships

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 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.

Different patterns may be more applicable to different projects, however. A strongly traditional project attempting to minimise absolute memory uses would place lots effort into **MEMORY BUDGETS** and **MEMORY TRACKING**, while an extreme project just trying to make sure a small program didn't crash too often would focus on **MEMORY PERFORMANCE ASSESSMENTS**, **PLUGGING THE LEAKS**, and **EXHAUSTION TESTING**.

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 for a particular project. The consequence section of each pattern describes how that pattern affects the forces you need to consider in applying the pattern.

There are a large number of other process patterns for software development, many of which are collected in the *Pattern Languages of Program Design* book series.

Memory Budget Pattern

A.k.a. Memory Costings

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

- You're working on a project where memory is limited.
- The project will fail if its memory requirements exceed the limits.
- You have several different components or tasks using memory
- Different individuals or teams may be responsible for each.
- Saving memory costs effort – it's easier to let someone else do it!
- Unnecessary optimisation wastes 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 (or even the entire project, including the hardware) unsaleable.

You could have everyone involved concentrate on minimising the system's memory requirements. This should certainly reduce the risk of the system becoming too big, but this scorched earth approach has its own risks – if you focus solely on keeping memory low, you'll have to accept trade-offs elsewhere such as poor time performance, difficult-to-use interfaces or large amounts of developer effort.

More importantly, some of the components of the system will provide more opportunities for saving memory 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?

Making decisions about where to save memory becomes more complicated when you also have to decide who should save memory. In projects with multiple developers or multiple teams, everyone likes to believe that the problem they are working on is unique, and harder than everyone else's problem. Since it takes time and effort to reduce memory use its only human to treat memory use as someone else's problem and hope that someone else will do the work. 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.

No one can predict the future, so a memory budget will never be correct. A quick 'back of the envelope' budget will be only roughly accurate, but even a more formal and more detailed memory budget will become less accurate as the project progresses. For this reason, a memory budget needs to be a living document that is kept up-to-date as the project progresses. Your should refer to it while doing **MEMORY TRACKING** during development, and during the **MEMORY PERFORMANCE ASSESSMENT** later in the project; revising the budget as you get better information about component's memory consumption, and then using further assessment to validate the new budget. You can apply a **MEMORY LIMIT** to each component in the finished system to ensure that they will never exceed their budgeted allocation. 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 [Brooks 1982]). 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 lose; if it turns out under budget then the budget will have been 'wrong' – so those doing the budget may loose credibility.

Implementation

Producing an accurate memory budget (and the subsequent **MEMORY TRACKING**) for a serious 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 memory budgets expends effort that could be better spent elsewhere. In an informal environment, with less emphasis on up-front design, developers may be actively hostile to a full-scale formal memory budget.

So 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. If these simple calculations suggest that memory will be tight then is it worth spending the effort to put together a more formal memory budget.

1. Which resources should you budget? Most systems have various different kinds of memory; with different constraints on each. Here are some possibilities:

- Main memory usage, including stack memory and any system overheads. Main memory requirements can often be analysed further, as follows.
- Global memory accessed by each component of the system.
- Heap space for each individual component or process (if components' memory space is limited).
- Control stack space for each thread or process.
- Operating System overheads (buffers, environment space, etc).
- Read-only memory space, for systems with code and data in ROM
- Secondary storage space, such as disks or flash RAM
- Total memory usage including RAM, main memory, and any code or data paged or swapped onto secondary storage.

It's generally easier to budget ROM usage than RAM. ROM allocation is constant, so you can budget a single figure for each component, and 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 ALLOCATION**.

Optimising a system's consumption of one of these resources will often be at the expense of the others. 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.

2. Enforcing Budgets in Software. Some environments provide memory use monitors or resource limits, which you can use to enforce memory budgets. The **MEMORY LIMIT** pattern describes how you can implement these limits yourself. You can use memory limits to enforce the budgeted 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.

3. Budgeting Variable Memory Requirements. The simplest approach to budgeting is to estimate the worst case memory use of each component and add them together, but this result is quite pessimistic. Some kinds of systems do require such a conservative approach (medical and process control software, for example), but many systems only exercise a few components at any one time. A digital diary, for example, will have only a few applications running concurrently. But even in less critical applications, different components' memory use is not independent. For example, the peak memory use of a wireless web browser is likely to coincide with the peak memory use of the network driver and web page cache.

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 requirements for each worst-case scenario. You can then estimate the total memory requirements for each scenario (by summing the peak or average usage for each component as appropriate) and negotiate a budget so that each scenario's total is less than the memory available to the system.

4. Dealing with Uncertainty. Software development in the real world is unpredictable. Often, it turns out to be just too difficult or too expensive to reduce every component's memory requirements to its 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.

To address this, ensure that there is some slack in the budget: a memory overdraft fund. The amount of memory to set aside depends on how uncertain the initial estimates are; typical overdraft allocations would be between 5% and 20% of the total budget. 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.

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 (**APPLICATION SWITCHING**), 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 [Palm 2000]. 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 1: Palm Pilot 3.0 Memory Budget

Known Uses

The OS/360 project included overdrafts as part of their budgets [Brooks75]. 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 is extremely limited. Based on a Memory Budget, the team devised compression and sharing techniques, and negotiated Featurectomy with their clients. Second, 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. Most versions of UNIX allow 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 — these examples are discussed further in the **MEMORY LIMIT** pattern.

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** uses post-hoc analysis to identify memory use hot spots and remove them. You can back up any of these approaches by enforcing **MEMORY LIMITS** at runtime.

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. 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.

The **SMALL ARCHITECTURE** pattern, and the other architectural patterns that follow it, describe how you can ensure each component in a system takes responsibility for its own memory use, and thus is more likely to meet its budget.

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

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

References

[Auer + Beck 96] Ken Auer and Kent Beck. Lazy Optimization: Patterns for Efficient Smalltalk Programming. Chapter 2 in *Patterns Languages of Program Design 2.* John M. Vlissides, James O. Coplien and Norm L. Kerth, editors. Addison-Wesley, 1996.

[Brooks 82] Frederick P. Brooks Jr. *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley, 1982.

[Flanders&Swan] The laws of thermodynamics. The Drop of (Another Hat). Parlophone.

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

[Palm 2000] Palm Inc. *Palm OS SDK Reference.* Palm Inc. Santa Clara, California. 2000.