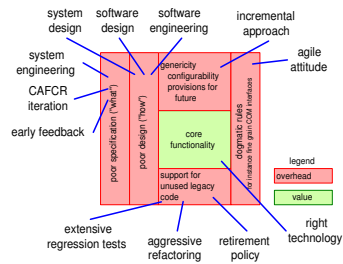


Exploration of the bloating of software

-



Gerrit Muller

Embedded Systems Institute

Den Dolech 2 (Laplace Building 0.10) P.O. Box 513, 5600 MB Eindhoven The Netherlands

gerrit.muller@embeddedsystems.nl

Abstract

Present-day products contain one order of magnitude more software code than is actually needed. The causes of this bloating are explored. If we are able to reduce the bloating significantly, then the product creation process is simplified tremendously. Potential handles to attack the bloating are discussed.

Distribution

This article or presentation is written as part of the Gaudí project. The Gaudí project philosophy is to improve by obtaining frequent feedback. Frequent feedback is pursued by an open creation process. This document is published as intermediate or nearly mature version to get feedback. Further distribution is allowed as long as the document remains complete and unchanged.

All Gaudí documents are available at:
<http://www.gaudisite.nl/>

version: 1.2

status: finished

February 10, 2011

1 Introduction

Bloating is one of the main causes of the *software crisis*. Bloating is the unnecessary growth of code. The really needed amount of code to solve a problem is often an order of magnitude less than the actual solution is using. Most SW based products contain an order of magnitude more software than is required. The cause of this excessive amount of software is explored in section 2 and 3.

The overall aspects of bloating are devastating: increased development, test and maintenance costs, degraded performance, increased hardware costs, loss of overview, et cetera.

2 Module level bloating

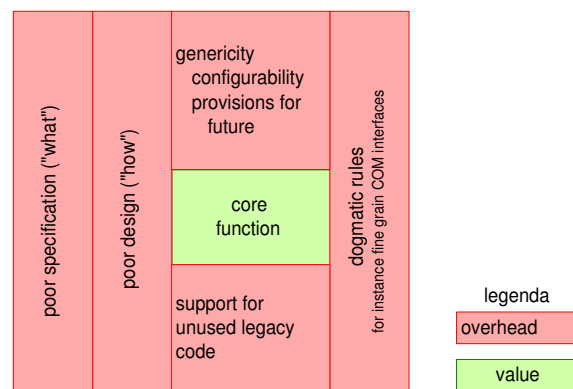


Figure 1: Exploring bloating

Figure 1 shows a number of causes for bloating. The specification of what need to be made is often wrong: too much functionality, wrong functionality, personal hobbyhorses, repair for previous poor specifications, et cetera. The main cause is insufficient understanding of the application, the customer needs and concerns, in other words insufficient understanding of the **why** behind the specification.

The design is the next source of bloating: ineffective design choices increase the code size. For example dynamic allocation is used, where the context allows for static allocation (dynamic uncertainty is added and need to be coped with, without adding value) or static allocation is used in a dynamic context (which results in dynamics to be added in an unnatural way, benefits of statics are not harvested, while a lot of complexity is added to cope with the dynamics). Insufficient design causes also a lot of bloating, for instance lots of duplicated functionality. Generic core functionality should have been factored out during design (but read the remarks about generic solutions below, factoring out requires know-how and skills).

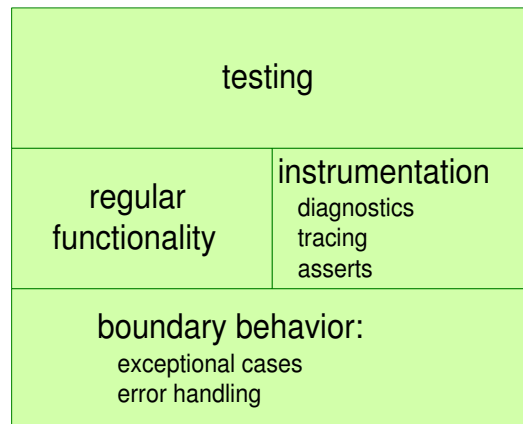
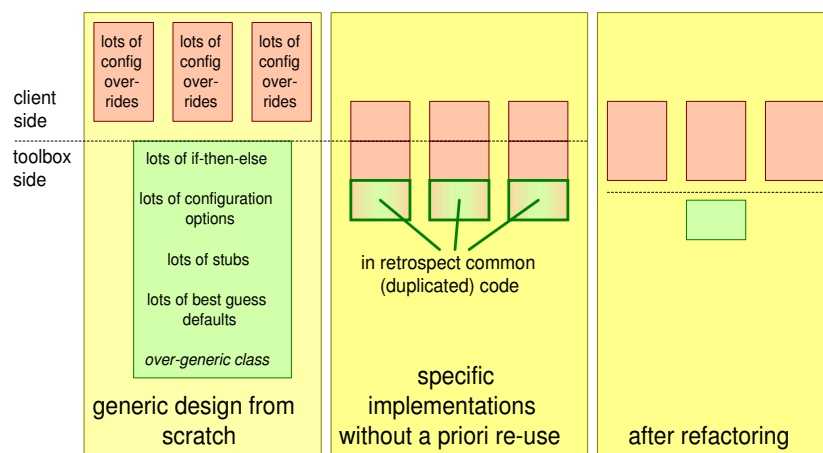


Figure 2: Necessary functionality is more than the intended regular function

Note that the core functionality in the center is all the required functionality to obtain a well behaved product. This means that it includes much more than the *regular* functionality, as shown in figure 2. This includes the boundary behavior (worst case situations, exceptions), instrumentation for development (tracing, debugging support, assertions, et cetera) and testing functionality. Note that all causes of bloating result in bloating of all these categories of regular functionality.

The drive towards generic solutions is often counterproductive. Figure 3 shows an actual example of part of the Medical Imaging system [7], which used a platform based reuse strategy. The reuse vision create a significant counterproductive drive towards generic solutions.



"Real-life" example: redesigned Tool super-class and descendants, ca 1994

Figure 3: The danger of being generic: bloating

The first implementation of a "Tool" class was over-generic. It contained lots of *if-then-else*, *configuration options*, *stubs for application specific extensions*, and lots of *best guess defaults*. As a consequence the client code based on this generic class contained lots of *configuration settings* and *overrides of predefined functions*.

The programmers were challenged to write the same functionality specific, which resulted in significantly less code. In the 3 specific instances of this functionality the shared functionality became visible. This shared functionality was factored out, decreasing maintenance and supporting new applications.

The next source of added overhead is caused by the dogmatic application of architecture rules. For instance the rule that components always communicate via COM. Such a rule might be very applicable for coarse grain components, but can ruin a fine grain design.

The last item which increases the code size is the accumulation of *unused* code. This happens slowly. In first instance the team is not aware of the fact that part of the functionality is not used anymore. Much later nobody knows what the effect will be if the unused code is removed. The motto becomes "if it ain't broke, don't fix", which results in an ever growing legacy of dead code.

3 Bloating causes more bloating

The bloating starts at low level. Via copy/paste modify existing bloating is propagated to new parts of the system. Figure 4 shows what happens with low level copy paste activities. An existing module is reused via copy-paste. The bad parts of the code are copied as well, which means that we now have the bad code twice in the repository. The new module has to do perform some new functionality, which means that new code, with its own bloating problems, is added. However in the copied code some unused code is not removed, while the bad code causes problems. These problems are solved by work-arounds.

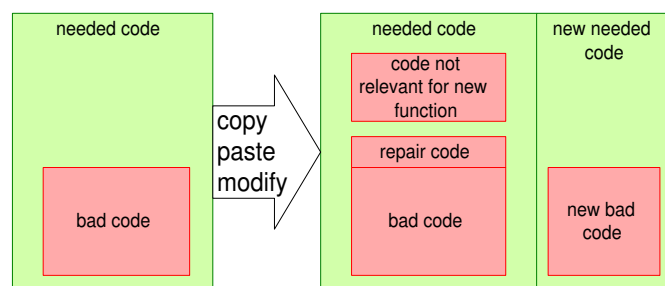


Figure 4: Shit propagation via copy paste

All together the new module is much worse bloated than the old module: shit propagation and amplification.

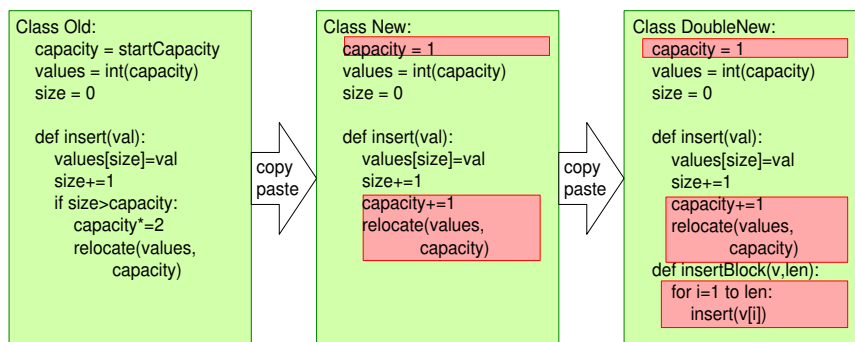


Figure 5: Example of shit propagation

An example of such shit propagation is shown in figure 5. An original module, with a locally embedded dynamic array pattern is copied in a new class. The original capacity doubling strategy is replaced by an incremental increase of the array. The original way of working with a *size* and a *capacity* has become obsolete, but it is not removed. The result is that the new class contains useless code, as well as uses more run time resources than strictly needed.

This poor implementation is itself again copied. Some new functionality is added, in this example a block insert. The block insert is implemented as repeated single inserts. Not only is the obsolete capacity structure still present, on top of that a very inefficient insertion is implemented, where for every element a complete re-allocate is performed.

This type of quality degradation can be found in many places in software repositories.

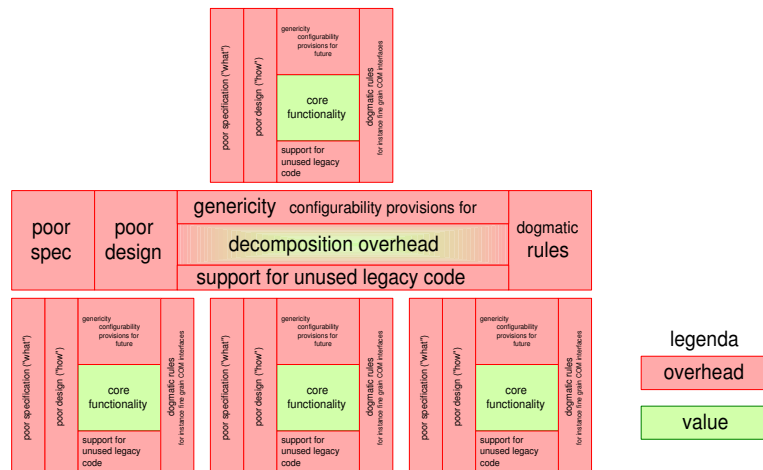


Figure 6: Bloating causes more bloating

One of the bloating problems is that bloating causes more bloating, as shown in figure 6. Software engineering principles force us to decompose large modules in smaller modules. "Good" modules are somewhere between 100 and 1000 lines of code. So where non-bloated functionality fits in one module, the bloated version is too large and needs to be decomposed in smaller modules. This decomposition adds some interfacing overhead. Unfortunately the same causes of overhead also apply to this decomposition overhead, which means again additional code.

All this additional code does not only cost additional development, test and maintenance effort, it also has run time costs: CPU and memory usage. In other words the system performance degrades, in some cases also with an order of magnitude. When the resulting system performance is unacceptable then repair actions are needed. The most common repair actions involve the creation of even more code: memory pools, caches, and shortcuts for critical functions. This is shown in figure 7.

Bloating causes performance and resource problems.
 Solution: special measures:
 memory pools, shortcuts, ...

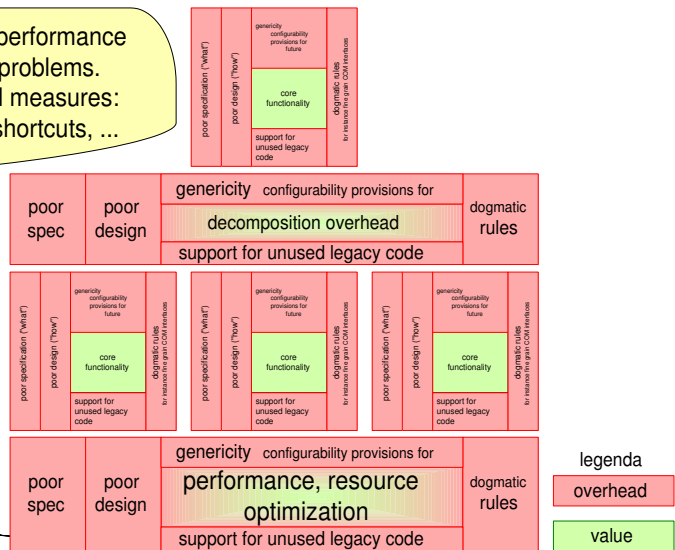


Figure 7: causes even more bloating...

4 What if we are able to reduce the bloating?

Lets assume that we are able to reduce the code size with a factor 5, in other words we can make an equivalent product with only 20% of the code size. Such a reduction would have a tremendous impact on the creation and the life-cycle afterwards of the product. Figure 8 shows some of the consequences.

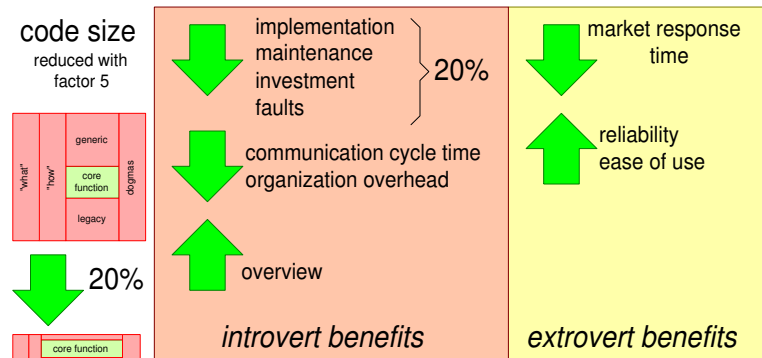


Figure 8: What if we remove half of the bloating?

The immediate consequence is that all parameters which are in first approximation proportional with the code size, will be reduced with the same factor. Imagine the impact of having 5 times less faults on the reliability or on the time needed for integration!

The creation crew and the maintenance crew decrease also proportional, which eases the communication tremendously. The organization also becomes much simpler and more direct. The housing demands are smaller, the crew fits in a smaller location. Figure 9 shows the relation between crew size, organization and housing.

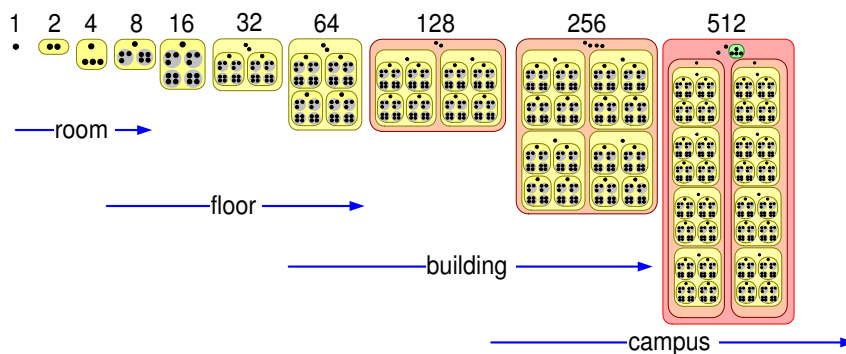


Figure 9: Impact of size on organization, location, process

Of course a reduction with a factor 5 is a tremendous challenge, a plan to attack the bloating is discussed in section 5. To achieve such an improvement the estimated overhead (circa 90% of the code) has to be reduced with a factor 8, and the core code has to be reduced with about 15% at the same time.

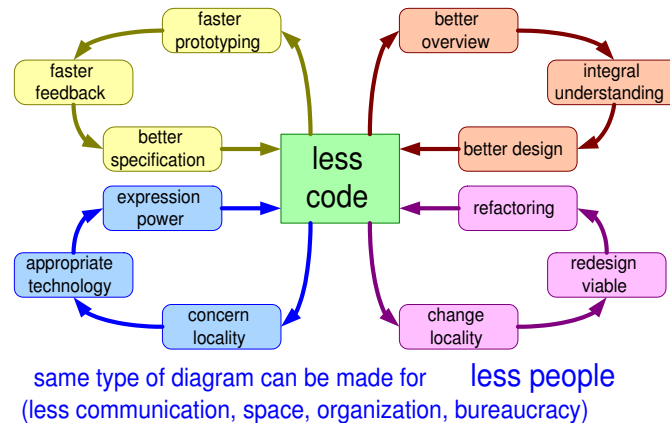


Figure 10: Anti bloating multiplier

If we are able to reverse the trend of bloating, an anti bloating multiplier effect will help us, as shown in figure 10. Less code helps in many ways to reduce the code even more: less code enables faster prototyping, which helps to get early feedback, which in turn improves the specification, and a better specification reduces the amount of code! Similar circular effects are obtained via the use of *right* technology, via refactoring and through improved overview.

The same multiplier effect is also present when we are able to reduce the crew size. Less people means easier communication, less distance, less need for bureaucratic control, less organizational overhead, all of them again reducing the amount of people needed!

5 How to attack the bloating?

The bloating must be attacked by coping with all the different causes of bloating as discussed in section 2. Figure 11 summarizes all different approaches that can be used to attack these different causes.

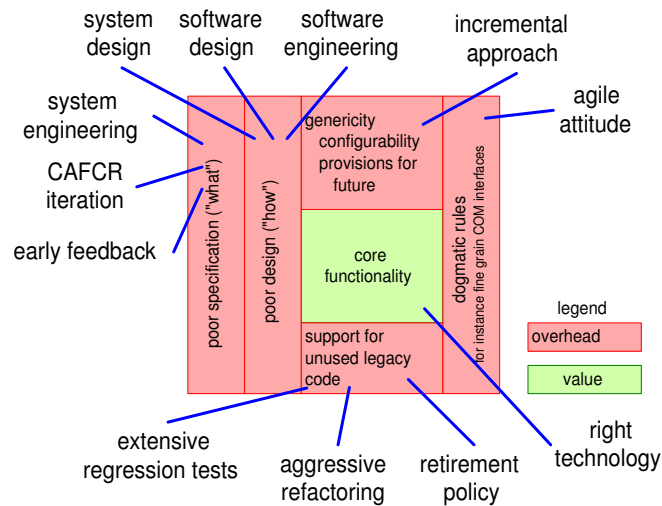


Figure 11: How to reduce bloating

5.1 Improving the specification

The systems engineering discipline is a matured discipline, for instance in the military and aero space domain, see for instance: [3] and [5]. Deploying methods and checklists from this discipline can help to improve specifications.

A major cause of poor specifications is late feedback, both from the customer side as well as from the technical cost and feasibility side. All modern product creation processes stress the importance of early feedback or an incremental approach, see [2], [1] and [4].

In [8] the CAFCR model is introduced as a means for architectural reasoning. From specification point of view it is important that the specification in the *Functional* view fits in the context of the *Customer objectives*, *Application*, *Conceptual* and *Realization* views. The architectural reasoning method is based upon fast iteration over the views and the different levels of abstraction.

5.2 Improving the design

One of the frequent design pitfalls is the dominance of a single decomposition. The consequence is that many other design dimensions are insufficiently taken into

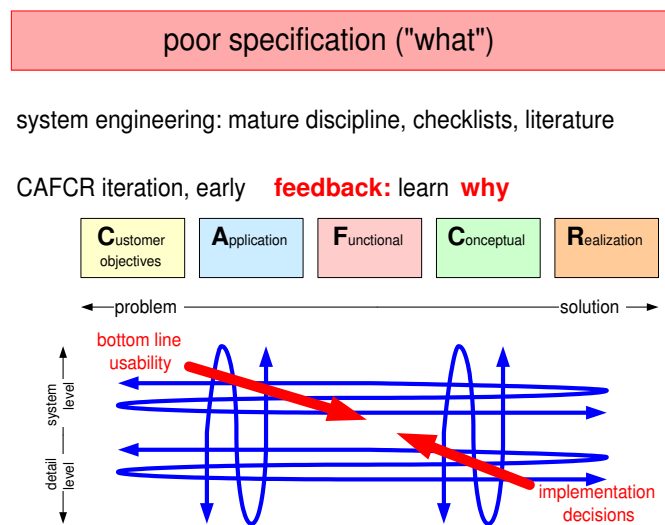


Figure 12: Improving the specification

account. The architectural reasoning method as described in [8] emphasizes the need for multiple views and methods in order to cope with many relevant design dimensions.

Figure 13 provides an overview of the architectural reasoning method based on the CAFCR model. Core to the deployment of the method is the availability of a rich collection of submethods, such that for each problem an approach is available, or at least that inspiration can be obtained from this rich set.

System design, software design and software engineering are closely related disciplines. System design can be tackled by means of CAFCR, as mentioned above. Software design requires sufficient conceptual skills: determining the concepts to be used: which generic functionality can (must) be factored out, where are specific solutions required. Finally good software engineering practices (naming conventions, tools, configuration management, et cetera) help to avoid commonly known mistakes. Trivial misnaming mistakes may cause lots of bloating, due to not recognizing concepts or structures.

5.3 Avoiding the genericity trap

Many software developers and architects love to create powerful and generic solutions. A truly powerful and generic solution can indeed be marvelous. Unfortunately these type of solutions often emerge after a lot of hard work and many trials. The mistake made by many of us is that we try to invent this ideal solution out of nothing, while the problem and solution know how is still rudimentary.

To avoid this genericity trap frequent feedback is essential. Understanding of

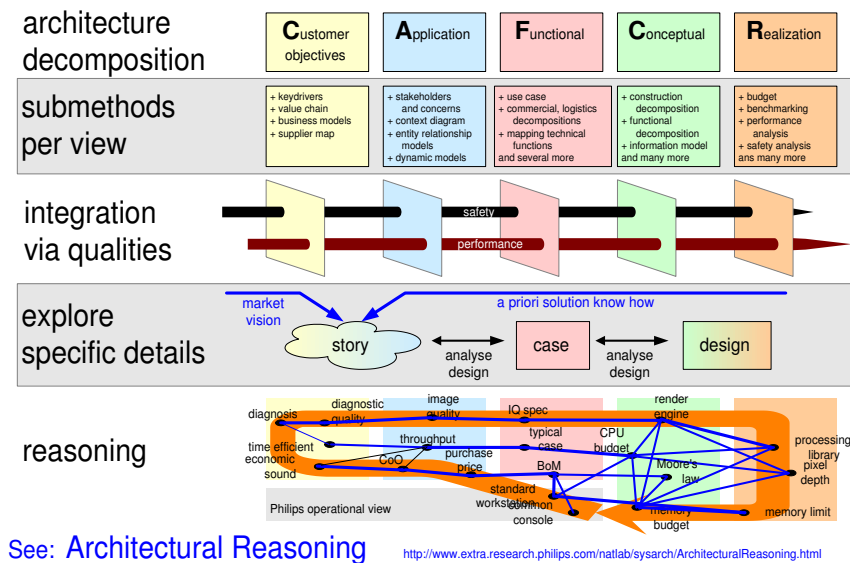


Figure 13: Use multiple views and methods

the problem as well as the solution is key to being effective. Learning via feedback is a quick way of building up this understanding. Waterfall methods all suffer from late feedback, see figure 14 for a visualization of the influence of feedback frequency on project elapsed time.

A more practical way to obtain more powerful and generic solutions is to start with learning. In practice after 3 initial implementations (often with some copy/paste based reuse), sufficient know how is available to factor out the generic part, see figure 15

5.4 Match solution technology with problem

The size of the functionality itself can often reduced by using the appropriate technology for the specific type of problem. Figure 16 shows some different types of technologies and the potential technology choice which can reduce the amount of code required to tackle the problem.

For user interface prototyping dedicated user interface or application generators are valuable tools. Many non hard real time problems of all kind of natures (textual, algorithmic, networking) can be expressed in high level problem terms by high level languages such as Python. When programming in Python much less code is required for all kinds of solution technology oriented needs.

For small hard real time or performance critical functions (for instance audio or image processing, or motion control) straightforward hand optimization is sometimes the most effective. All kinds of high level constructs in this problem domain trigger

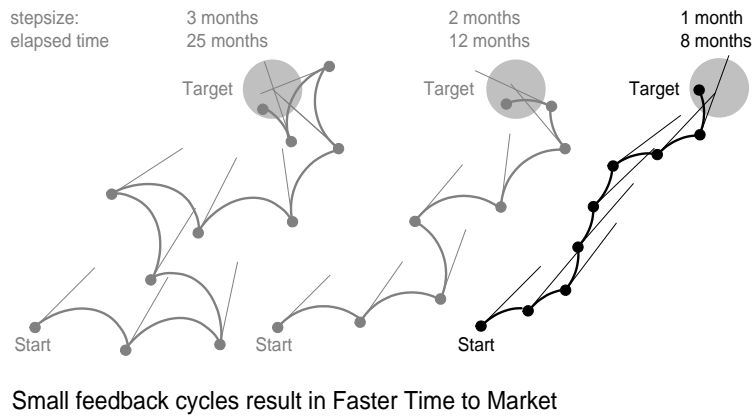
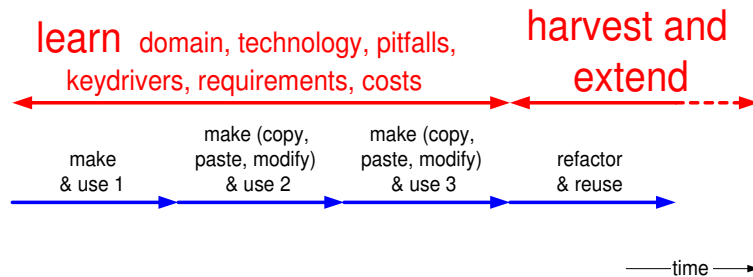


Figure 14: Feedback (3)



heuristic: use 3 times before factoring out the generic parts

Figure 15: Lesson learned about reuse

the bloating process due to all additional measures needed to meet performance or timing needs.

Highly repeatable problems, with small variations, can be addressed by specialized generators. Development of dedicated toolkits for this class of problems is often highly efficient in terms of amount of code and cost.

5.5 Agility instead of dogmatism

An agile attitude is needed to avoid dogmatic application of all kinds of architecture rules. In [?] recommendations are given to achieve a light weight architecture.

Figure 17 from [?] shows the tension between the different objectives of an architecture. *Flexibility* requires agility, while *manageability* requires more control through architecture rules. Organizational growth or maturity often involves an


UI prototyping:	GUI editor/generator
non hard real time textual, algorithmic, networking:	 Python
small hard real-time or extremely performance critical	hand optimized
highly repeatable problem	dedicated generator tools

Figure 16: Examples of "right" technology choices

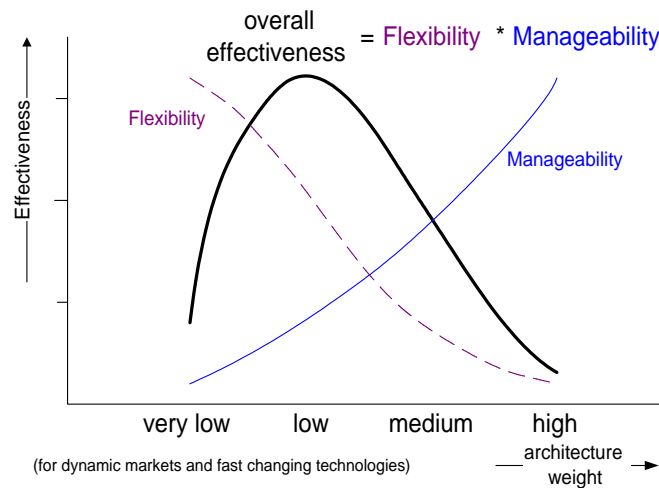


Figure 17: Keep the architecture weight low

increase of *manageability*, which can backfire if this translates in dogmatism.

5.6 Reduce unused code

The first step in removing unused code is to have a retirement policy: how is retirement communicated, how long are old features supported, support for obsolescence detection et cetera.

When features are retired a cleanup of the associated code is required. Quite some drive is required to actually do this, an aggressive refactoring mentality is quite helpful to achieve this.

As described in the problem analysis the cleanup is often not done out of fear: what might happen somewhere else in the code if we remove this? Extensive regression test suites help to detect this kind of problems and help to remove the

support for unused legacy code

retirement policy	make explicit what can not be used anymore
aggressive refactoring	cleanup
extensive regression tests	reduce fear reduce surprises

Figure 18: Reduce unused code

fear of cleanup.

6 Acknowledgements

Wim Mosterman reminded me of the “shit propagation” effect, which causes a significant amount of bloating. Nick Maclaren pointed out that factoring out generic functionality during design (not during programming!) is an effective anti-bloat measure. Tom Hoogenboom for providing feedback.

References

- [1] B.W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, May 1988.
- [2] Thomas Gilb. Evolutionary object management. <http://www.gilb.com/Download/EVOART95.ZIP>, 1996.
- [3] INCOSE. International council on systems engineering. <http://www.incose.org/toc.html>, 1999. INCOSE publishes many interesting articles about systems engineering.
- [4] Philippe B. Kruchten. A rational development process. *Crosstalk* 9, pages 11–16, July 1996.
- [5] James N. Martin. *Systems Engineering Guidebook*. CRC Press, Boca Raton, Florida, 1996.
- [6] Gerrit Muller. The system architecture homepage. <http://www.gaudisite.nl/index.html>, 1999.
- [7] Gerrit Muller. Case study: Medical imaging; from toolbox to product to platform. <http://www.gaudisite.nl/MedicalImagingPaper.pdf>, 2000.
- [8] Gerrit Muller. Architectural reasoning explained. <http://www.gaudisite.nl/ArchitecturalReasoningBook.pdf>, 2002.

History

Version: 1.2, date: July 7, 2003 changed by: Gerrit Muller

- added factoring out generic functionality during design
- changed status to “finished”

Version: 1.1, date: June 4, 2003 changed by: Gerrit Muller

- added "shit propagation"

Version: 1.0, date: June 4, 2003 changed by: Gerrit Muller

- updated bloating visualization figures
- added reuse heuristic
- added text
- added reduce unused diagram
- changed status to draft

Version: 0.2, date: June 2, 2003 changed by: Gerrit Muller

- added figure "reduce what"
- added figures how to reduce

Version: 0.1, date: May 28, 2003 changed by: Gerrit Muller

- added abstract
- added stub sections "what if...", "how to attack..."

Version: 0, date: May 16, 2003 changed by: Gerrit Muller

- Created, no changelog yet