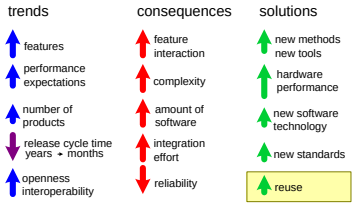
Software Reuse; Caught between strategic importance and practical feasibility

-



Gerrit Muller
Buskerud University College
Frogs vei 41 P.O. Box 235, NO-3603 Kongsberg Norway
gaudisite@gmail.com

Abstract

Worldwide the belief is shared that software reuse is needed to cope with the ever increasing amount of software. Software reuse is one part of addressing the amount of software, which is often overhyped and underestimated. Reuse of software is discussed via 8 statements, addressing: the need for reuse, the technical and organizational challenges, integration issues, evolution, reuse of know how, focus on the bussiness and customer and validation.

version: 1.0 status: concept October 20, 2017

1 Introduction

Many good reasons exists to deploy a reuse strategy for product creation, see figure 1. This list, the result of a brainstorm, can be extended with more objectives, but this list is already sufficiently attractive to consider a reuse strategy.

+ reduced time to market
+ reduced cost per function
+ improved quality
+ improved reliability
+ easier diversity management
+ employees only have to understand one base system
+ improved predictability
+ larger purchasing power
+ means to consolidate knowledge
+ increase added value
+ enables parallel developments of multiple products
+ free feature propagation

Figure 1: Why reuse: many valid objectives

Reuse is deployed already in many product development centers. Brainstorming with architects involved in such developements about their experiences gives a very mixed picture, see figure 2 for the bad versus the good experiences.

bad	good
longer time to market	reduced time to market
high investments	reduced investment
lots of maintenance	reduced (shared) maintenance cost
poor quality	improved quality
poor reliability	improved reliability
diversity is opposed	easier diversity management
lot of know how required	understanding of one base system
predictable too late	improved predictability
dependability	larger purchasing power
knowledge dilution	means to consolidate knowledge
lack of market focus	increase added value
interference	enables parallel developments
but integration required	free feature propagation

Figure 2: Experiences with reuse, from counterproductive to effective

Analysis of the positive experiences show that successful applications of a

reuse strategy share one or more of the following characteristics: *homogeneous domain*, *hardware dominated* or *limited scope*. Figure 3 shows a number of examples.

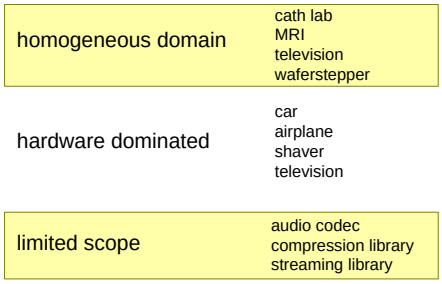


Figure 3: Succesful examples of reuse

Reuse strategies can work successfully for a long time and then suddenly run into problems. Figure 4 shows the limitations of successful reuse strategies.

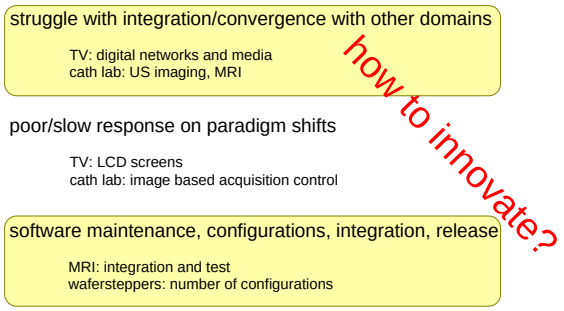


Figure 4: Limits of successful reuse

The main problem with successful reuse strategies is that they work efficient as long as the external conditions evolve slowly. However breakthrough events don't fit well in the ongoing work which results in a poor response.

About half of this article reuses previous Gaudí articles by copy, paste and sometimes modify. Articles used are: [6] [5] [7] [1] [9] [4]

2 Statements about reuse

Reuse of software is a mixture of believe, hype, hope and solution of a set of problems. To stimulate the discussion about reuse a set of statements is postulated in figure 5 and 6 about reuse.

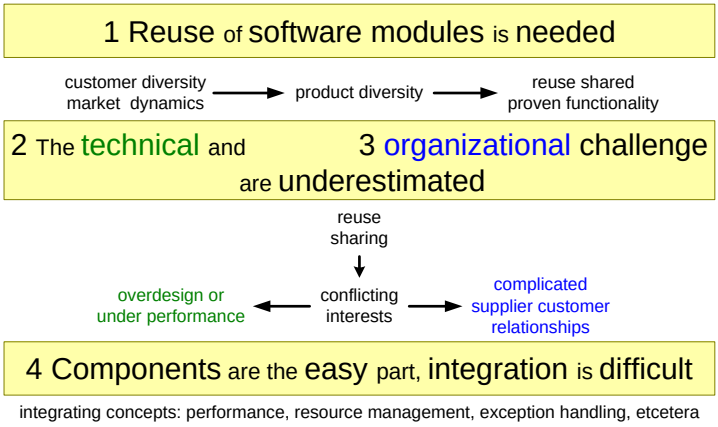


Figure 5: Reuse statements

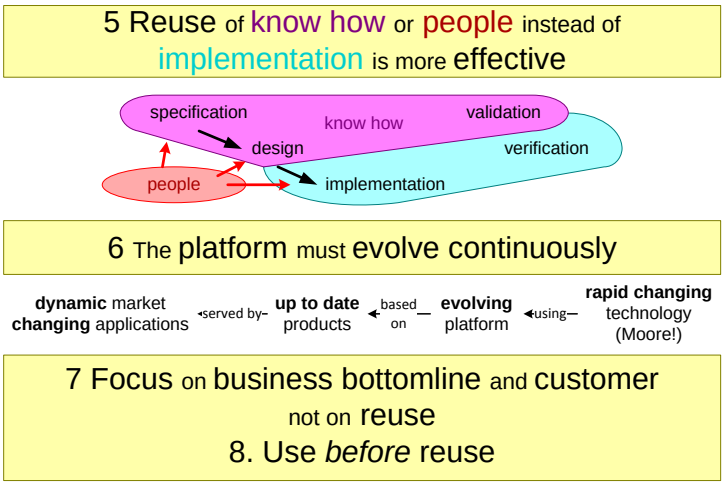


Figure 6: Reuse statements continued

3 Software reuse is needed

The trends in the market are towards more products, each with more feature and higher performance expectations. Products are expected to work seamlessly with other products, even with new products and formats which did not yet exist when the product was conceived: openness and interoperability is required. All of these expectations have to be fulfilled in less and less time, product creation life cycles have decreased from years to months.

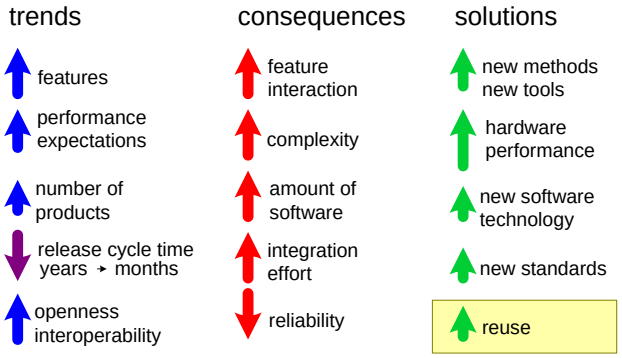


Figure 7: Reuse is needed ... as part of the solution

Figure 7 show these trends in the market in the left hand column, where the length of the arrow indicate the relative increase or decrease.

The consequence of the market trends for product creation are that more and more features start to interact and that the complexity increases. This is reflected in a string growth in the amount of software in products. The integration effort increases also. The combination of these factors threaten the reliability, products which simply cease operating have become a fact of life.

To accomodate these trend multiple solutions need to be applied concurrently, as shown in the right hand column. New methods and tools are needed, which fit in this fast evolving, connected world. The fast developments of the hardware (Moore's law) help significantly in following the expectations in the market. New software technology, increasing the abstraction level used by programmers, increases the productivity and reduces complexity. New standards reduce the interoperability issues.

Reuse of software modules potentially decreases the creation effort, enables focus on the required feautures and increases the quality if the modules have been proven.

4 The technical challenge

How to determine which functionality is generic and which functionality must be implemented specific? Practical experience learns that this is a crucial question. Most attempts to create a platform of reusable components fail due to the creation of overgeneric components.

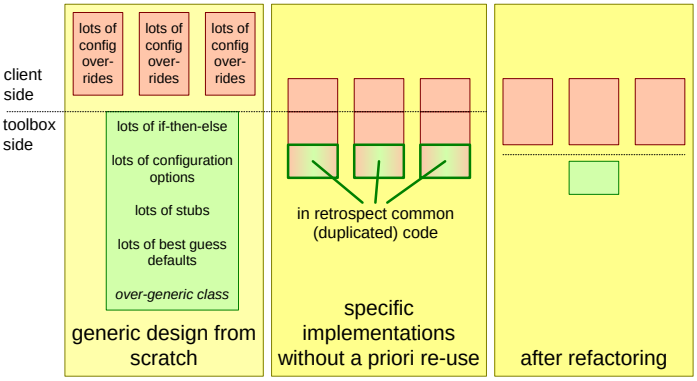


Figure 8: The danger of being generic: bloating

Figure 8 show an actual example of part of the Medical Imaging system [3], which used a platform based reuse strategy. The first implementation of a ”Tool” class was overgeneric. 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 settins* 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.

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. Figure 9 shows a number of causes for bloating.

One of the bloating problems is that bloating causes more bloating, as shown in figure 10. 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 unbloated 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

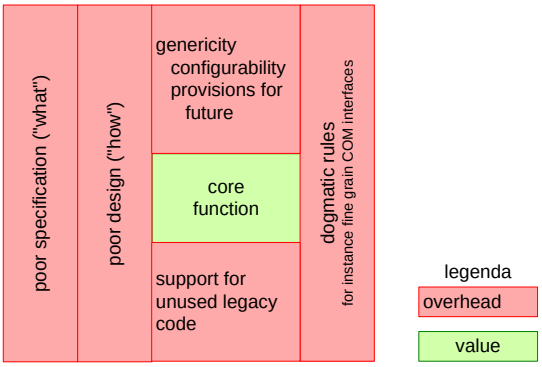


Figure 9: Exploring bloating

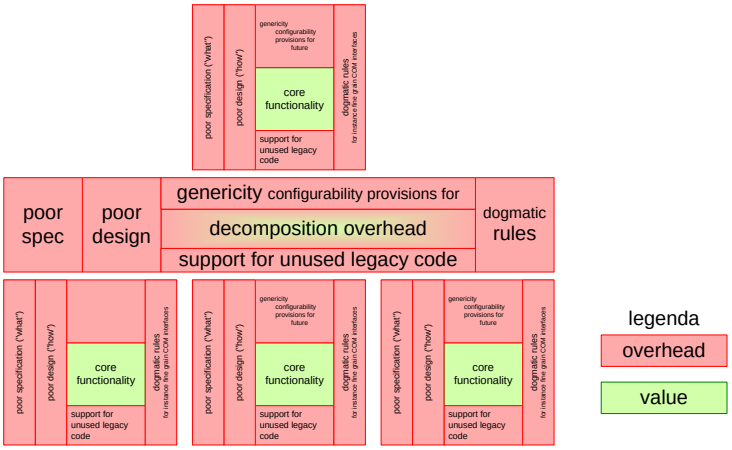


Figure 10: Bloating causes more bloating

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.

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

Reuse should not trigger such a bloating process, because the bloating will undo all the reuse benefits.

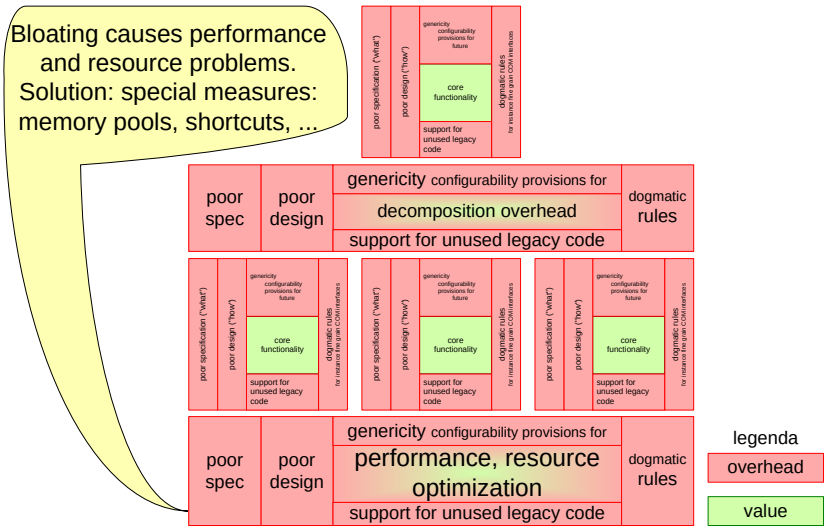


Figure 11: causes even more bloating...

5 The organizational challenge

The operational organization of the product creation process for a portfolio or family of products used to be a simple hierarchy: portfolio, product family, product, subsystem, module. The 3 main product creation roles are operational (project management), technical (architecture) and commercial (marketing, product management). These 3 roles are present at the different hierarchical levels, although the commercial role is often not needed for the internal subsystems and modules. Figure 12

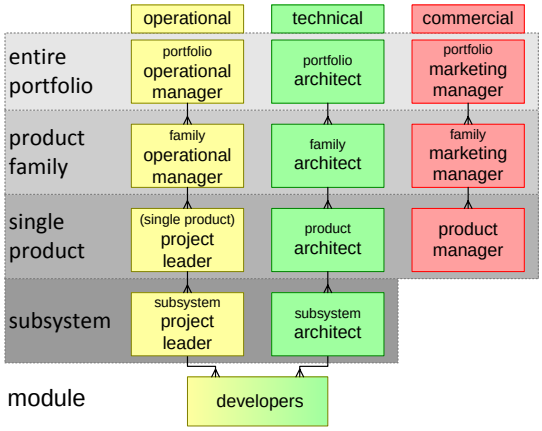


Figure 12: Conventional operational organization

The introduction of reuse has a big impact on this hierarchy in the operational organziation of the PCP. Figure 13 shows the organization after the addition of a shared platform of shared components. The platform project leader reports directly to the operational manager of the product family. His other core team members also report directly to the family counter part: platform architect to family architect, platform manager to family marketing manager. The supplier relationship is that the platform delivers to the product, in other words the product creation is the customer of the platform creation.

Figure 14 focuses on the tension which created by the sharing of a single platform creation by multiple product creations. Conflicting interests with respect to platform functionality or performance cannot be solved by the individual product creation teams, but is propagated to the family level. At family level the policy is set, which is executed by the platform creation. The platform team has to disappoint one or more of its customers in favor of another customer.

The same problems happens with external suppliers, where the supplier has to satisfy multiple customers. The main difference is that in such a supplier customer relationship economic rules apply, where a dissatisfied customer will change from supplier. The threshold to change from supplier in platform driven organizations is

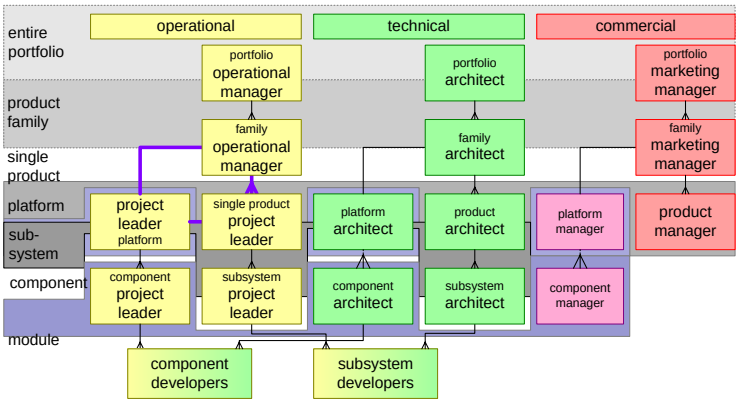


Figure 13: Modified operational organization

very high, disrupting the normal economic control system.

The ultimate consequence is less commitment and satisfaction in product creation (receiving the blame, without being in control) plus a lot of political hassle where people try to achieve their objectives *despite* the organization.

Figure 13 contains a few other pecularities. First of all commercial roles appear for internal products. At the moment that the organization complexity increases with internal suppliers and customers also internal ”commercial” functions appear, such as account managers. They act at the interfaces between the groups, inventarizing requirements and promoting solutions.

Another pecularity is the existence of both a family architect as well as platform architect. The family architect has a wider scope than the platform architect, with more application content. The platform architect is more focused at the technology/solution side: how to provide the required infrastructure. Note that both architects must have a lot of overlap: the platform architect must understand the application context, the family architect must understand the solution space.

One of the frequent occuring mistakes is the *inversion of control*, where the platform team starts to determine the family policy. The platform creation must enable the family policy, but should not determine this policy.

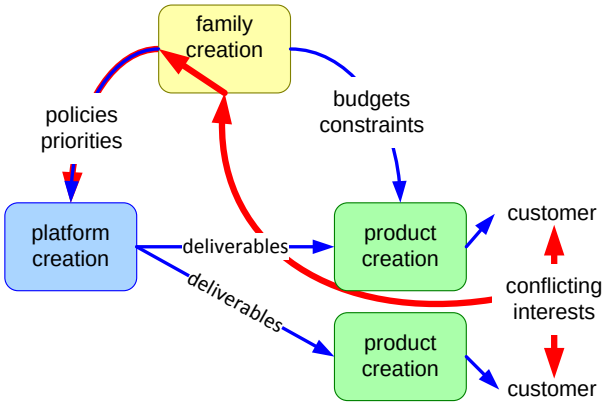


Figure 14: Conflicting interests of customers escalate to family level, have impact on platform, product creation teams benefit or suffer from the top down induced policy

6 Integration

Many people expect the architect to decompose, as mentioned in the explanation of ”guiding how”, while integration is severely underestimated, see figure 15. In most development projects the integration is a traumatic experience. It is a challenge for the architect to make a design which enables a smooth integration.

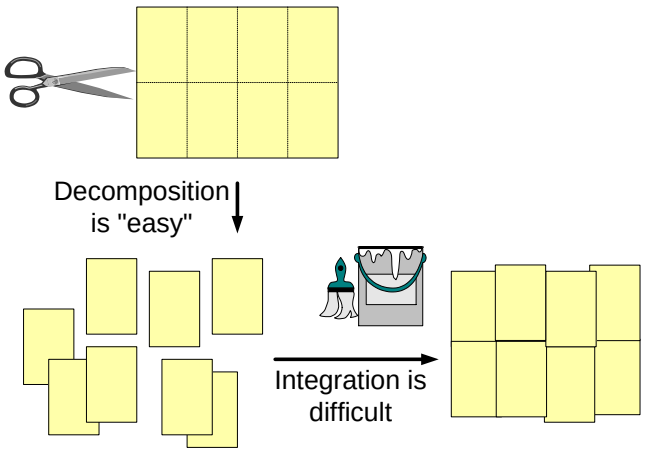


Figure 15: Decomposition is easy, integration is difficult

Projects run without (visible) problems during the decomposition phases. All components builders are happily designing, making and testing their component. When the integration begins problems become visible. Figure 16 visualizes this process. The invisible problems cause a significant delay[1].

Combining existing software packages is mostly difficult due to ”architectural mismatches”. Different design approaches with respect to exception handling, resource management, control hierarchy, configuration management et cetera, which prohibit straightforward merging. The solution is adding lots of code, in the form of wrappers, translators and so on, while this additional code adds complexity, it does not add any end-user value.

Performance and resource usage are most often far from optimal after a merger.

Amazingly many people start worrying about duplication of functionality when merging, while this is the least of a problem in practice. This concern is the cause of reuse initiatives, which address the wrong (non-existing) problem: duplication, while the serious architectural problems are not addressed.

Creating the solution is a collective effort of many designers and engineers. The architect is mostly guiding the implementation, the actual work is done by the

[1]This is also known as the *95% ready syndrome*, when the project members declare to at 95%, then actually more than half of the work still needs to be done.

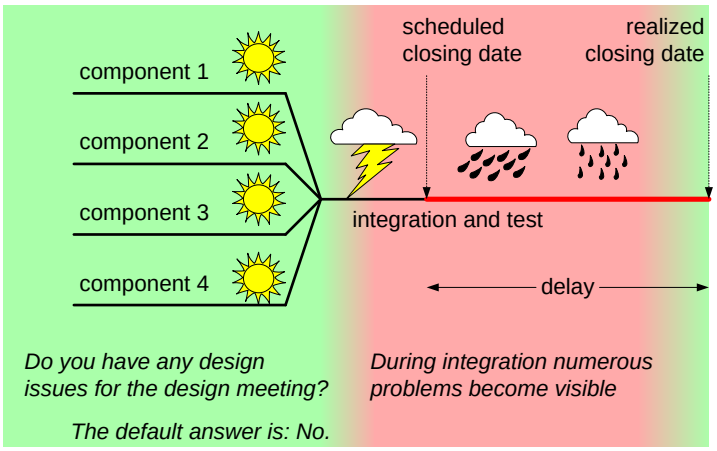


Figure 16: Integration problems show up late during the project, as a complete surprise

designers and engineers. Guiding the implementation is done by providing guidelines and high level designs for many different viewpoints. Figure 18 shows some of the frequently occurring viewpoints for guiding the implementation. Note that many people think that the major task of the architect is to define **the** decomposition and to define and manage the interfaces of this decomposition. Figure 18 shows that architecting involves many more aspects and especially the integrating concepts are crucial to get working products.

The deliverables of a platform development can range from requirement specifications, to designs to complete implementations. Figure 19 shows a blueprint of a full blown platform.

The blueprint shows a superset of what can be part of the platform. Figure 19 shows different variants, subsets, which can be used as a platform.

The type **A** platform consists of concepts and small building blocks. The integration of all blocks has to be done by the product creators.

Type **B** platforms deliver the generic parts, for instance the computing infrastructure. Note that this includes the infrastructure related parts of the architecture guidelines.

Type **C** is an application oriented platform. This type of platform is much more pre-integrated and pre-tested.

At the bottom-right the platforms are positioned in the integration space, see [1].

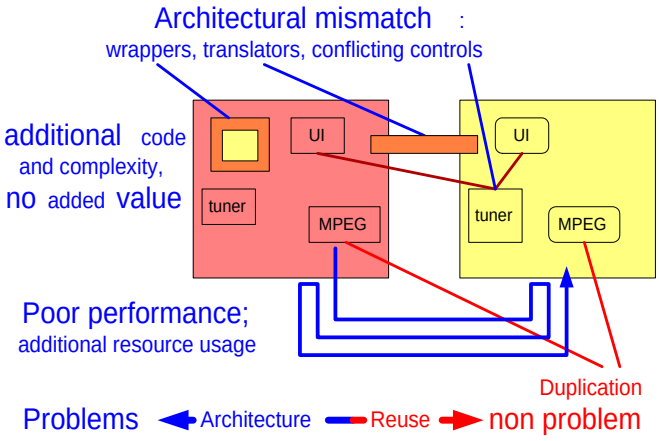


Figure 17: Integration of components from different sources is difficult due to the architectural mismatch

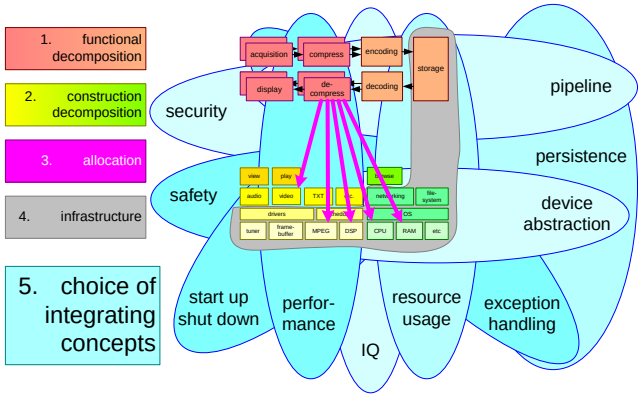


Figure 18: Integrating concepts

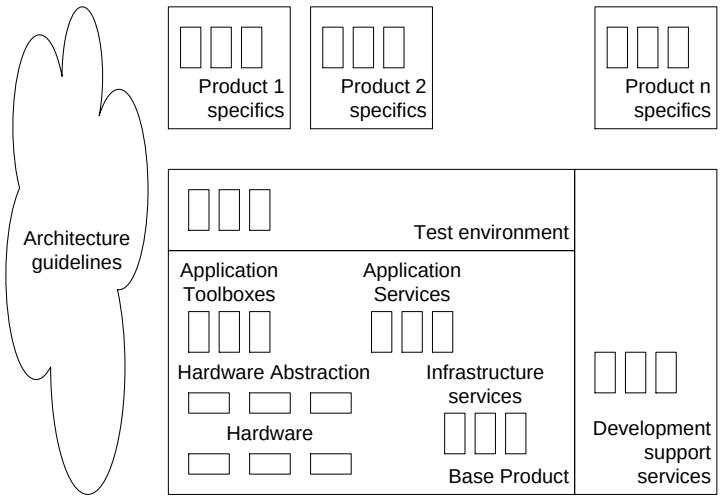


Figure 19: Platform block diagram

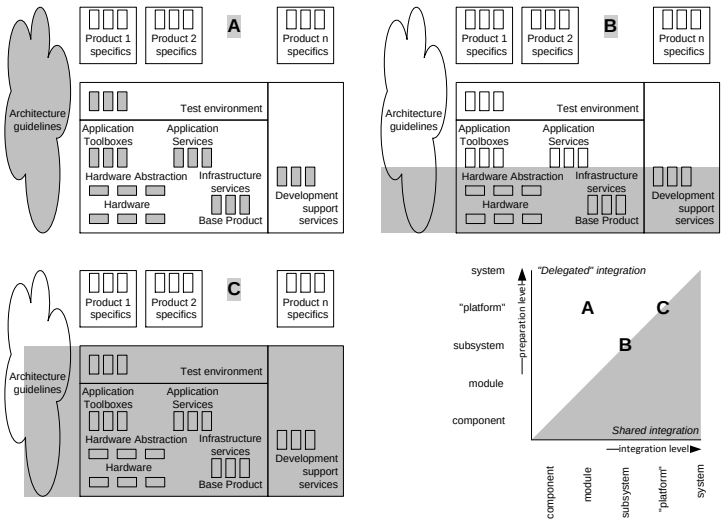


Figure 20: Platform types

7 Evolution

A common pitfall is that managers as well as engineers expect a platform to be stable; once the platform is created only a limited maintenance is needed. Figure 21 explains why this is a myth. A platform is build using technology that itself is changing very fast (Moore's law again). At the other hand a platform serves a dynamic fast changing market, see for example [6]. In other worls it is a miracle if a platform is stable, when both the supplying as well as the consuming side are not stable at all.

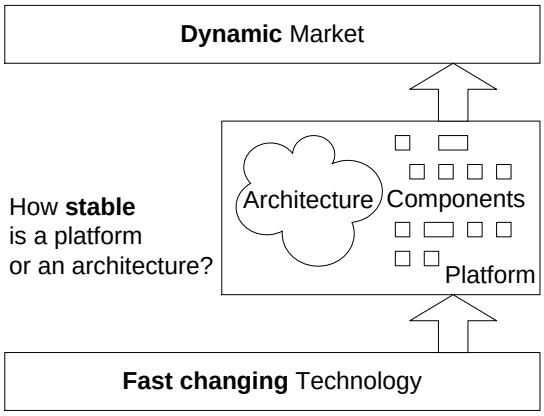


Figure 21: The outside world is dynamic

The evolution of a platform is illustrated in figure 22 by showing the change in the Easyvision [3] platform in the period 1991-1996. It is clearly visible that every generation doubles the amount of code, while at the same time half of the existing code base is touched by changes.

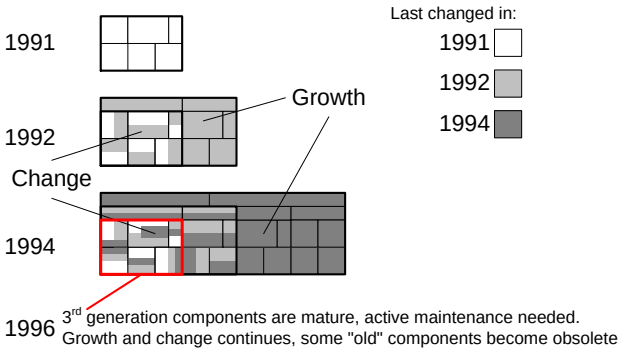


Figure 22: Platform evolution (Easyvision 1991-1996)

8 Reuse of know how

The CAFCR model [8] uses 5 views to look at an architecture. Most discussions about reuse are concerned about the reuse of implementation, working code. Implementation is part of the realization view. However reuse of the other views is more easy and can be quite beneficial.

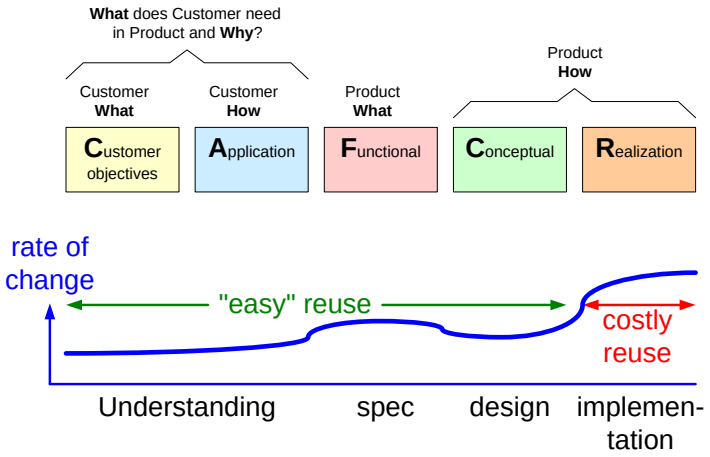


Figure 23: Reuse in CAFCR perspective

Figure 23 shows the CAFCR model at the top. Below the rate of change is shown for the different views. The rate of change in the implementation view is very high. All changes from the other views accumulate here, and on top of that the fast change of the technology is added.

Reusing an implementation is like shooting for a fast moving target. The actual benefits might never be harvested, due to obsolescence of the used implementation. The understanding of the customer is a quite valuable resource. Due to the conservative nature of most humans the half-life of this know how is quite long.

The understanding of the customer is translated into specifications. These specifications have a shorter half-life, due to the competition and the technology developments. Nevertheless reuse of specifications, especially the generic parts, can be very rewarding.

The conceptual view contains the more stable insights of the design. The CAFCR model on purpose factors out the concepts, because concepts are reused by nature.

9 Focus on business bottomline and customer

One of the big risks of reuse is that the focus of the organization and the people shifts from solutions and value for the customer to the internals of the product design, the technology used in the generic components.

This change of focus can be understood by the following simplified model of a business. The business process for an organization which creates and builds systems consisting of hardware and software is decomposed in 4 main processes as shown in figure 24.

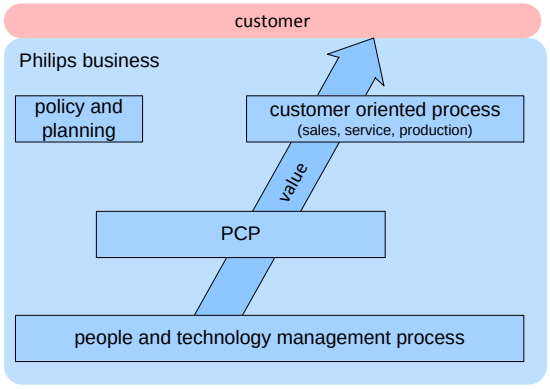


Figure 24: Simplified decomposition of the business in 4 main processes

The decomposition in 4 main processes leaves out all connecting supporting and other processes. The function of the 4 main processes is:

Customer Oriented Process This process performs in repetitive mode all direct interaction with the customer. This primary process is the cashflow generating part of the enterprise. All other processes only spend money.

Product Creation Process This Process feeds the Customer Oriented Process with new products. This process ensures the continuity of the enterprise by creating products which enables the primary process to generate cashflow tomorrow as well.

People and Technology Management Process Here the main assets of the company are managed: the know how and skills residing in people.

Policy and Planning Process This process is future oriented, not constrained by short term goals, it is defining the future direction of the company by means of roadmaps. These roadmaps give direction to the Product Creation Process and the People and Technology Management Process. For the medium term these roadmaps are transformed in budgets and plans, which are committal for all stakeholders.

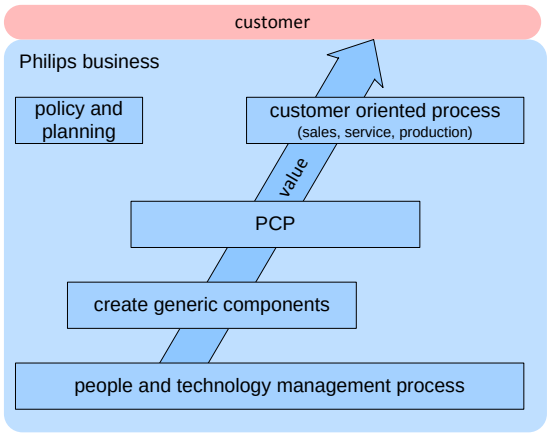


Figure 25: Modified Process Decomposition

The simplified process description given in figure 24 assumes that product creation processes for multiple products are more or less independent. When generic developments are factored out for strategic reasons an additional process is required to visualize this. Figure 25 shows the modified process decomposition (still simplified of course) including this additional process "Generic Something Creation Process".

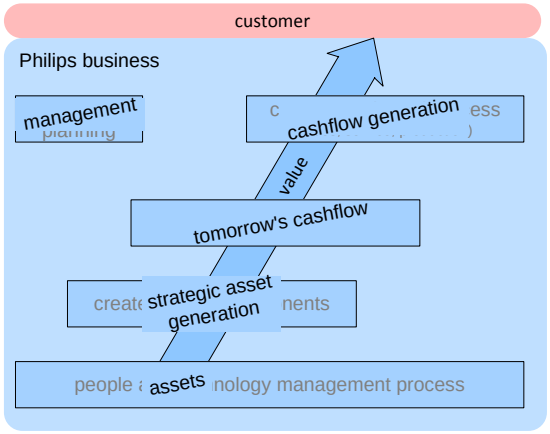


Figure 26: Financial Viewpoint on Process Decomposition

Figure 26 shows these processes from the financial point of view. From financial point of view the purpose of this additional process is the generation of strategic assets. These assets are used by the product generation process to enable tomorrow's cashflow.

The consequence of this additional process is an lengthening of the value chain

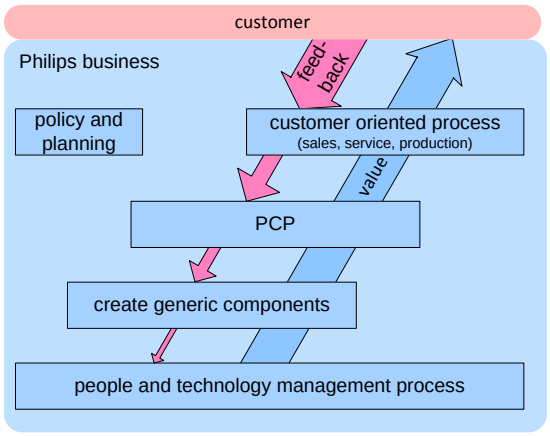


Figure 27: Feedback flow: loss of customer understanding!

and consequently a longer feedback chain as well. This is shown in figure 27. The increased length of the feedback chain is a significant threat for generic developments.

Many different models for the development of generic things are in use. An important differentiating characteristic is the driving force, which often directly relates to the de facto organization structure. The main flavors of driving forces are shown in figure 28.

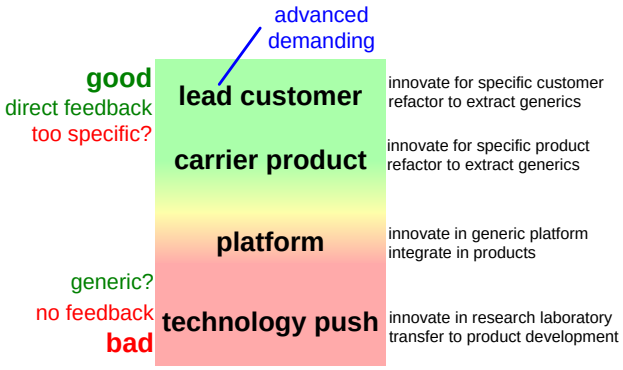


Figure 28: Models for SW reuse

9.1 Lead Customer

The lead customer as driving force guarantees a direct feedback path from an actual customer. Due to the importance of feedback this is a very significant advantage.

The main disadvantages of this approach are that the outcome of such a development often needs a lot of work to make it reusable as a generic product. The focus is on the functionality and performance, while many of the quality aspects are secondary in the beginning. Also the requirements of this lead customer can be rather customer specific, with a low value for other customer.

9.2 Carrier Product

The combination of a generic development with one of the product developments also shortens the feedback cycle, although it is not as direct as with the lead customer. Combination with a normal product development will result in a better balance between performance and functionality focus and quality aspects. Disadvantage can be that the operational team takes full ownership for the product (which is good!), while giving the generic development second priority, which from family point of view is unwanted.

In larger product families the different charters of the product teams creates a political tension. Especially in immature or power oriented cultures this can lead to horrible counterproductive political games.

Lead customer driven product development, where the product is at the same time the carrier for the platform combines the benefits of the lead customer and the carrier product approach. In my experience this is the most effective approach of generic developments. A prerequisite for success is an open and result driven culture to preempt any political game mentioned before.

9.3 Platform

In maturing product families the generic developments are often decoupled from the product developments. In products where integration plays a major role (which are nearly all products) the generic developments are pre-integrated into a platform or base product, which is released to be used by the product developments.

The benefit of this approach is separation of concerns and decoupling of products and platforms in smaller manageable units. Both benefits are also the main weakness of such a model, as a consequence the feedback loop is stretched to a dangerous length. At the same time the time from feature/technology to market increases, see figure 29.

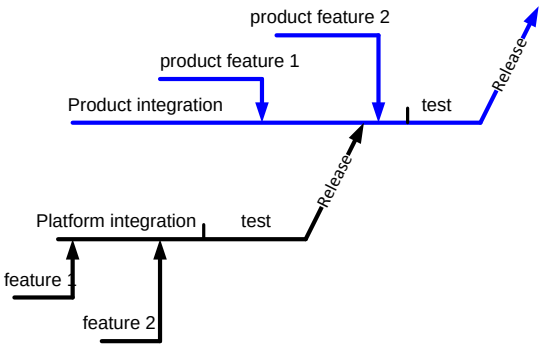


Figure 29: The introduction of a new feature as part of a platform causes an additional latency in the introduction to the market.

10 Use before reuse

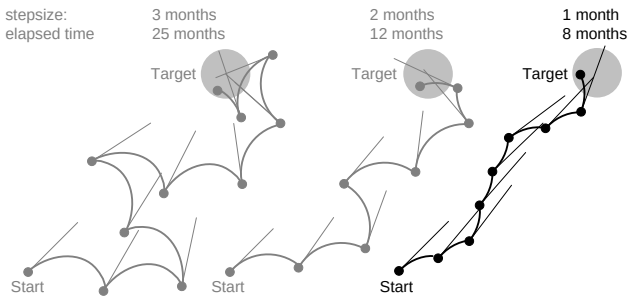


Figure 30: Feedback (3)

Understanding of 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 30 for a visualization of the influence of feedback frequency on project elapsed time.

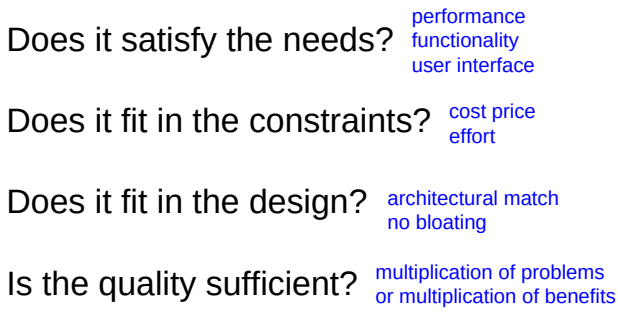


Figure 31: Use of software modules enables validation before Reuse

References

[1] Gerrit Muller. Product families and generic aspects. `http://www.gaudisite.nl/GenericDevelopmentsPaper.pdf`, 1999.

[2] Gerrit Muller. The system architecture homepage. `http://www.gaudisite.nl/index.html`, 1999.

[3] Gerrit Muller. Case study: Medical imaging; from toolbox to product to platform. http://www.gaudisite.nl/MedicalImagingPaper.pdf, 2000.

[4] Gerrit Muller. Process decomposition of a business. http://www.gaudisite.nl/ProcessDecompositionOfBusinessPaper.pdf, 2000.

[5] Gerrit Muller. From legacy to state-of-the-art; architectural refactoring. http://www.gaudisite.nl/ArchitecturalRefactoringPaper.pdf, 2001.

[6] Gerrit Muller. Light weight architectures; the way of the future? http://www.gaudisite.nl/info/LightWeightArchitecting.info.html, 2001.

[7] Gerrit Muller. The system architect; meddler or savior? http://www.gaudisite.nl/MeddlerOrSaviorPaper.pdf, 2001.

[8] Gerrit Muller. Architectural reasoning explained. http://www.gaudisite.nl/ArchitecturalReasoningBook.pdf, 2002.

[9] Gerrit Muller. The importance of system architecting for development. http://www.gaudisite.nl/ImportanceOfSAforDevelopmentPaper.pdf, 2002.

History

Version: 1.0, date: March 19, 2003 changed by: Gerrit Muller

- added 2 figures about platform types and integration
- changed status to concept

Version: 0, date: March 4, 2003 changed by: Gerrit Muller

- Created, no changelog yet
- About half of this article reuses previous Gaudí articles by copy, paste and sometimes modify.