

# Composable Architectures

logo  
TBD

Gerrit Muller

University of South-Eastern Norway-NISE

Hasbergsvei 36 P.O. Box 235, NO-3603 Kongsberg Norway

[gaudisite@gmail.com](mailto:gaudisite@gmail.com)

## Abstract

Composable architectures are used to create product families and individual products of a product family. This book bundles articles addressing several concerns and approaches with respect to composing products in a composable architecture.

### **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/>

# Contents

<b>Introduction</b>	<b>xi</b>
<b>1 How to Create a Manageable Platform Architecture?</b>	<b>1</b>
1.1 Introduction . . . . .	1
1.2 Case: Medical Imaging Workstation . . . . .	2
1.2.1 Product Context . . . . .	2
1.2.2 Historic Phases . . . . .	4
1.2.3 Basic Application and Toolboxes . . . . .	4
1.2.4 Medical Imaging X-Ray . . . . .	5
1.2.5 Second Concurrent Product: Medical Imaging CT/MR . .	8
1.2.6 Towards Workflow . . . . .	10
1.3 Architecture . . . . .	13
1.4 Platform . . . . .	17
1.5 The Time Dimension . . . . .	22
1.6 Process View . . . . .	27
1.6.1 Lead Customer . . . . .	30
1.6.2 Carrier Product . . . . .	31
1.6.3 Platform . . . . .	31
1.7 Market Driven . . . . .	32
1.8 Recommendations . . . . .	36
<b>2 A Method to Explore Synergy between Products</b>	<b>37</b>
2.1 Introduction . . . . .	37
2.2 Stepwise method to explore synergy opportunities . . . . .	38
2.2.1 Explore markets, customers, products and technologies . .	39
2.2.2 Share market and customer insights . . . . .	40
2.2.3 Identify product features and technology components . . .	41
2.2.4 Make maps . . . . .	42
2.2.5 Discuss value, synergy and (potential) conflicts . . . . .	43
2.2.6 Create long term and short term plan . . . . .	43
2.3 Example of synergy . . . . .	44

<b>3</b>	<b>Software Reuse; Caught between strategic importance and practical feasibility</b>	<b>45</b>
3.1	Introduction . . . . .	45
3.2	Statements about reuse . . . . .	48
3.3	Software reuse is needed . . . . .	49
3.4	The technical challenge . . . . .	50
3.5	The organizational challenge . . . . .	53
3.6	Integration . . . . .	56
3.7	Evolution . . . . .	60
3.8	Reuse of know how . . . . .	61
3.9	Focus on business bottomline and customer . . . . .	62
3.9.1	Lead Customer . . . . .	64
3.9.2	Carrier Product . . . . .	65
3.9.3	Platform . . . . .	65
3.10	Use before reuse . . . . .	67
<b>4</b>	<b>Aggregation Levels in Composable Architectures</b>	<b>68</b>
4.1	Problem description . . . . .	68
4.2	Views on Aggregation . . . . .	69
4.3	Documentation . . . . .	69
4.4	Source Code Management viewpoint . . . . .	70
4.5	Composition viewpoint . . . . .	72
4.5.1	Optimal granularity for composition . . . . .	73
4.6	Field Deployment viewpoint . . . . .	74
4.7	Integration and Test viewpoint . . . . .	74
4.8	Acknowledgements . . . . .	76
<b>5</b>	<b>From Legacy to State-of-the-art; Architectural Refactoring</b>	<b>77</b>
5.1	The problem . . . . .	77
5.1.1	Market trends . . . . .	77
5.1.2	Technology trends . . . . .	79
5.1.3	Example Digital Television . . . . .	80
5.2	Architectural Refactoring . . . . .	84
5.2.1	Prerequisites for effective architectural refactoring . . . . .	85
5.3	Conclusion . . . . .	90
5.4	Acknowledgements . . . . .	90
<b>6</b>	<b>Light Weight Architecture: the way of the future?</b>	<b>92</b>
6.1	Introduction . . . . .	92
6.2	Do the right things; The Dynamic Market . . . . .	94
6.3	Do the things right; Lessons from Practice . . . . .	102

6.4	The Weight of an Architecture; Architectural Chaos or Bureaucratic Control? . . . . .	106
6.5	Light weight how-to . . . . .	109
6.6	Summary . . . . .	111
6.7	Acknowledgements . . . . .	111
<b>7</b>	<b>Exploration of the bloating of software</b>	<b>112</b>
7.1	Introduction . . . . .	112
7.2	Module level bloating . . . . .	112
7.3	Bloating causes more bloating . . . . .	116
7.4	What if we are able to reduce the bloating? . . . . .	119
7.5	How to attack the bloating? . . . . .	121
7.5.1	Improving the specification . . . . .	121
7.5.2	Improving the design . . . . .	121
7.5.3	Avoiding the genericity trap . . . . .	122
7.5.4	Match solution technology with problem . . . . .	123
7.5.5	Agility instead of dogmatism . . . . .	124
7.5.6	Reduce unused code . . . . .	125
7.6	Acknowledgements . . . . .	127

# List of Figures

1.1	Outline of this paper . . . . .	2
1.2	Philips Medical Systems, schematic organization overview. . . . .	2
1.3	Generic drivers of Radiology Departments . . . . .	3
1.4	Phases of Medical Imaging . . . . .	4
1.5	Technology innovations by Common Viewing . . . . .	5
1.6	Idealized layering of SW toolboxes and Basic Application in september 1991 . . . . .	5
1.7	X-ray rooms from examination to reading around 1990 . . . . .	6
1.8	X-ray rooms from examination to reading, when Medical Imaging is applied as printserver . . . . .	6
1.9	Comparison of conventional <i>screen copy</i> based film and a film produced by Medical Imaging. This case is very favorable for the Medical Imaging approach, typical gain is 20% to 50%. . . . .	7
1.10	Idealized layers of the Medical Imaging R/F software in september 1992 . . . . .	7
1.11	Example of Multi Planar Reformating applied on the spine . . . . .	8
1.12	Example of CT and MR department, where Medical Imaging is deployed . . . . .	8
1.13	Idealized layers of the Medical Imaging software in june 1994 . . . . .	9
1.14	Radiology department as envisioned in 1996 . . . . .	10
1.15	Medical Imaging in health-care workflow perspective, as envisioned in 1996 . . . . .	11
1.16	Idealized layers of the Medical Imaging software in 1996 . . . . .	12
1.17	What is Architecture? . . . . .	13
1.18	What is Architecture? . . . . .	13
1.19	"Guiding How" by providing rules for: . . . . .	14
1.20	The Art of Architecting . . . . .	14
1.21	Architecting is much more than Decomposition . . . . .	15
1.22	The architecture description is by definition a flattened and poor representation of an actual architecture. . . . .	15
1.23	Drivers of Generic Developments . . . . .	17

1.24	What is a Platform? . . . . .	18
1.25	Platform Source Deliverables . . . . .	19
1.26	And now in More Detail... . . . .	19
1.27	Example of Platform Efficiency . . . . .	20
1.28	Purchased SW Requires Embedding . . . . .	20
1.29	Embedding Costs of Purchased SW . . . . .	21
1.30	Example of Embedding Problems . . . . .	21
1.31	Who is First: Platform or Product? . . . . .	22
1.32	Myth: Platforms are Stable . . . . .	23
1.33	The first time right? . . . . .	23
1.34	Feedback (3) . . . . .	24
1.35	Platform Evolution (Easyvision 1991-1996) . . . . .	24
1.36	Life-cycle Differences . . . . .	25
1.37	Reference model for health care automation . . . . .	26
1.38	Simplified decomposition of the business in 4 main processes . . . . .	27
1.39	Modified Process Decomposition . . . . .	28
1.40	Financial Viewpoint on Process Decomposition . . . . .	28
1.41	Feedback flow: loss of customer understanding! . . . . .	29
1.42	The introduction of a new feature as part of a platform causes an additional latency in the introduction to the market. . . . .	29
1.43	Sources of failure in platform developments . . . . .	30
1.44	Models for SW reuse . . . . .	30
1.45	The “CAFCR” model . . . . .	32
1.46	Five viewpoints for an architecture. The task of the architect is to integrate all these viewpoints, in order to get a <i>valuable, usable</i> and <i>feasible</i> product. . . . .	33
1.47	Example of Scoping of a Platform. . . . .	33
1.48	Example of the four key drivers in a motorway management system . . . . .	34
1.49	Example Thread of Reasoning from the Medical Imaging Workstation . . . . .	35
1.50	Summary of recommendations to manage platform architectures . . . . .	36
2.1	Types of synergy . . . . .	37
2.2	Approach to Platform Business Analysis . . . . .	38
2.3	Explore Markets, Customers, Products and Technologies . . . . .	39
2.4	Study one Customer and Product . . . . .	40
2.5	Work Flow Analysis for Different Customers/Applications . . . . .	40
2.6	Make Map of Customers and Market Segments . . . . .	41
2.7	Identify Product Features and Technology Components . . . . .	42
2.8	Mapping From Markets to Components . . . . .	42
2.9	Example Criteria for Determining Value . . . . .	43
2.10	Determine Value of Features . . . . .	43

2.11	Example of synergy between heterogeneous markets . . . . .	44
3.1	Why reuse: many valid objectives . . . . .	46
3.2	Experiences with reuse, from counterproductive to effective . . . .	47
3.3	Succesful examples of reuse . . . . .	47
3.4	Limits of successful reuse . . . . .	47
3.5	Reuse statements . . . . .	48
3.6	Reuse statements continued . . . . .	48
3.7	Reuse is needed ... as part of the solution . . . . .	49
3.8	The danger of being generic: bloating . . . . .	50
3.9	Exploring bloating . . . . .	51
3.10	Bloating causes more bloating . . . . .	51
3.11	causes even more bloating... . . . .	52
3.12	Conventional operational organization . . . . .	53
3.13	Modified operational organization . . . . .	54
3.14	Conflicting interests of customers escalate to family level, have im- pact on platform, product creation teams benefit or suffer from the top down induced policy . . . . .	55
3.15	Decomposition is easy, integration is difficult . . . . .	56
3.16	Integration problems show up late during the project, as a complete surprise . . . . .	57
3.17	Integration of components from different sources is difficult due to the architectural mismatch . . . . .	58
3.18	Integrating concepts . . . . .	58
3.19	Platform block diagram . . . . .	59
3.20	Platform types . . . . .	59
3.21	The outside world is dynamic . . . . .	60
3.22	Platform evolution (Easyvision 1991-1996) . . . . .	60
3.23	Reuse in CAFCR perspective . . . . .	61
3.24	Simplified decomposition of the business in 4 main processes . . .	62
3.25	Modified Process Decomposition . . . . .	63
3.26	Financial Viewpoint on Process Decomposition . . . . .	63
3.27	Feedback flow: loss of customer understanding! . . . . .	64
3.28	Models for SW reuse . . . . .	64
3.29	The introduction of a new feature as part of a platform causes an additional latency in the introduction to the market. . . . .	66
3.30	Feedback (3) . . . . .	67
3.31	Use of software modules enables validation before Reuse . . . . .	67
4.1	Venn diagram showing the overlap between Viewpoints on Aggre- gation Levels . . . . .	69
4.2	Visualization of documentation concerns . . . . .	70

4.3	The source code is stored in files in a repository. The unit of structuring is called a package. These source code aggregation levels get a more semantic meaning when being used. . . . .	71
4.4	Coarse versus Fine grained with respect to the number of connections and relations; 9 large Components with 18 Connections, 81 small Components with 648 Connections . . . . .	73
4.5	Integration and testing as function of size . . . . .	76
5.1	Today's Audio Video Consumer Products . . . . .	78
5.2	Trend: Convergence of separate worlds . . . . .	78
5.3	Integration and Diversity . . . . .	79
5.4	Today's Video Products . . . . .	79
5.5	Evolution of Video Products . . . . .	80
5.6	Distribution Scenario's . . . . .	81
5.7	Product Packaging Options . . . . .	82
5.8	Moore's law . . . . .	82
5.9	Problem: increasing SW size, decreasing reliability? . . . . .	82
5.10	The Holy Grail: Reuse . . . . .	83
5.11	Simplistic Architecting: Digital TV . . . . .	83
5.12	Available Code Assets . . . . .	83
5.13	Merge problems . . . . .	84
5.14	Solution: Architectural Refactoring . . . . .	85
5.15	Example of Refactoring Goals . . . . .	85
5.16	Architectural and Code refactoring . . . . .	86
5.17	Frequent feedback results in faster results and a shorter path to the result . . . . .	87
5.18	Myth: Platforms are Stable . . . . .	88
5.19	Platform Evolution (Easyvision 1991-1996) . . . . .	88
5.20	Example Long Term Vision . . . . .	89
5.21	Don't do . . . . .	90
5.22	Conclusion: Refactoring the Architecture is a must . . . . .	91
6.1	What is Architecture? . . . . .	92
6.2	Table of Contents . . . . .	93
6.3	Value chain . . . . .	94
6.4	Convergence . . . . .	95
6.5	Integration and Diversity . . . . .	95
6.6	Uncertainty (Dot.Com effect) . . . . .	96
6.7	Moore's law . . . . .	96
6.8	System Integrator Problem Space - Business . . . . .	97
6.9	System Integrator Problem Space - Technology . . . . .	97
6.10	System profile . . . . .	98



6.11	PS Technology solutions . . . . .	99
6.12	Partial Solution: Configurable Component Platform . . . . .	99
6.13	Exploring problem space and solution ingredients . . . . .	100
6.14	More than Architecture . . . . .	100
6.15	Conclusions Part 1A . . . . .	101
6.16	"Guiding How" by providing rules for: . . . . .	102
6.17	The Art of Architecting . . . . .	103
6.18	Architecting is much more than Decomposition . . . . .	104
6.19	Myth: Platforms are Stable . . . . .	104
6.20	The first time right? . . . . .	105
6.21	Example with different feedback cycles (3, 2, and 1 months) showing the time to market decrease with shorter feedback cycles . . . . .	105
6.22	Platform Evolution (Easyvision 1991-1996) . . . . .	105
6.23	Architecture Weight . . . . .	106
6.24	Scope and Impact . . . . .	106
6.25	Criteria for an Architecture . . . . .	107
6.26	Weight versus Effectiveness . . . . .	107
6.27	Conclusion Part 2 . . . . .	108
6.28	Light Weight How -To . . . . .	109
6.29	Minimize Rule Weight . . . . .	110
6.30	Summary . . . . .	111
7.1	Exploring bloating . . . . .	113
7.2	Necessary functionality is more than the intended regular function . . . . .	113
7.3	The danger of being generic: bloating . . . . .	114
7.4	Shit propagation via copy paste . . . . .	116
7.5	Example of shit propagation . . . . .	116
7.6	Bloating causes more bloating . . . . .	117
7.7	causes even more bloating... . . . .	118
7.8	What if we remove half of the bloating? . . . . .	119
7.9	Impact of size on organization, location, process . . . . .	119
7.10	Anti bloating multiplier . . . . .	120
7.11	How to reduce bloating . . . . .	121
7.12	Improving the specification . . . . .	122
7.13	Use multiple views and methods . . . . .	123
7.14	Feedback (3) . . . . .	124
7.15	Lesson learned about reuse . . . . .	124
7.16	Examples of "right" technology choices . . . . .	125
7.17	Keep the architecture weight low . . . . .	125
7.18	Reduce unused code . . . . .	126

# List of Tables

4.1	<i>Concerns per viewpoint . . . . .</i>	69
4.2	<i>Aggregation Levels or Entities per viewpoint . . . . .</i>	70
4.3	<i>Typical Sizes of SW for Aggregation Levels . . . . .</i>	72
4.4	<i>The relation between the number of components and the required number of architects, zero order model . . . . .</i>	74
4.5	<i>The relation between the number of components and the required number of architects, first order model . . . . .</i>	75
4.6	<i>Decomposition of Field Deployment granularity drivers . . . . .</i>	75

# Introduction

This book bundles the articles about Composable Architectures Research.

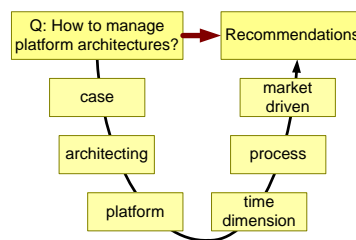
At this moment the book is in its early infancy. Most articles are updated based on feedback from readers and students. The most up to date version of the articles can always be found at [15]. The same information can be found here in presentation format.

Chapters can be read as autonomous units.



# Chapter 1

## How to Create a Manageable Platform Architecture?



### 1.1 Introduction

Most companies struggle with the development of functionality and components shared by multiple products. The strategy to share development costs of shared functionality and components is known under many different labels: re-use, product families, product lines, generic developments or platforms to name a few. We will use the term *platform* in this paper.

This paper is partially, about half, based on existing Gaudí material. We want to address the following question in this paper: “Q: How to manage platform architectures?”. Figure 1.1 shows the outline of this paper. We start by discussing an actual platform case that covers more than 10 years elapsed time. Next we explore *architecting* and *platforms*. We zoom in on the *time dimension*, the *process* and the need to be *market driven*. Finally we summarize by a means of a number of recommendations.

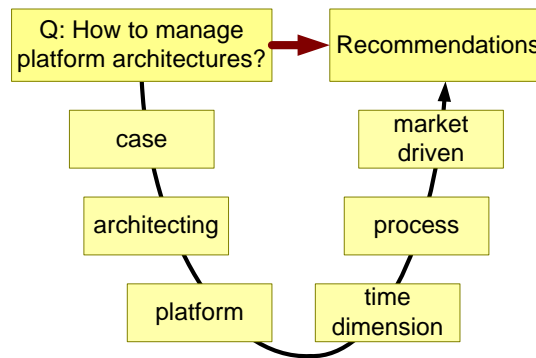


Figure 1.1: Outline of this paper

## 1.2 Case: Medical Imaging Workstation

The Medical Imaging workstation was an early large scale Object Oriented product. Originally intended to become a re-useable set of toolboxes, it evolved in a family of medical workstations and servers.

### 1.2.1 Product Context

Philips Medical Systems is a major player in the medical imaging market. The main competitors are GE and Siemens. The Product Creation focus of Philips Medical Systems is modality oriented, as shown in figure 1.2.

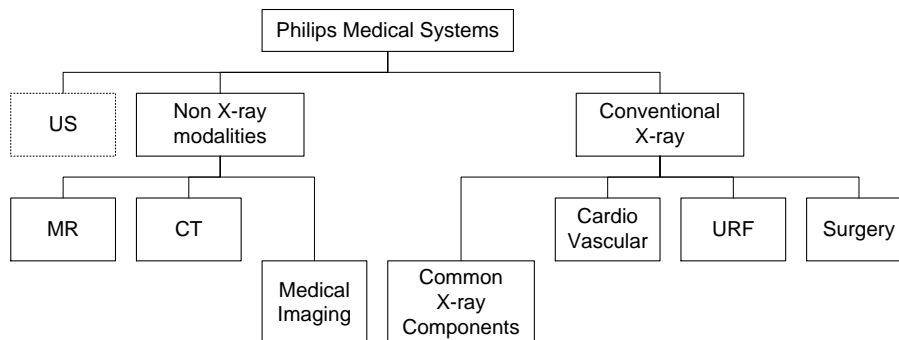


Figure 1.2: Philips Medical Systems, schematic organization overview.

The common technology in conventional X-ray systems is developed by component oriented business groups, which make generators, tubes, camera's, detectors, et cetera. The so-called "System-groups" have a more clinical focus, they create the clinical oriented systems on the basis of the common available components.

The non X-ray groups<sup>1</sup> mainly build large complex general purpose imaging equipment. The imaging principles in CT and MR are less direct, which means that an image reconstruction step is required after acquisition to form the viewable images. Ultra Sound (ATL) is acquired by Philips Medical Systems recently. It is not fully integrated in the organization. The main markets of Philips Medical Systems are radiology and cardiology, with a spin off to the surgery market.

Traditionally the radiologist makes and interprets images from the human body. A referring physician requests an examination, the radiologist responds with a report with his findings. Figure 1.3 shows a generic set of Radiology drivers.

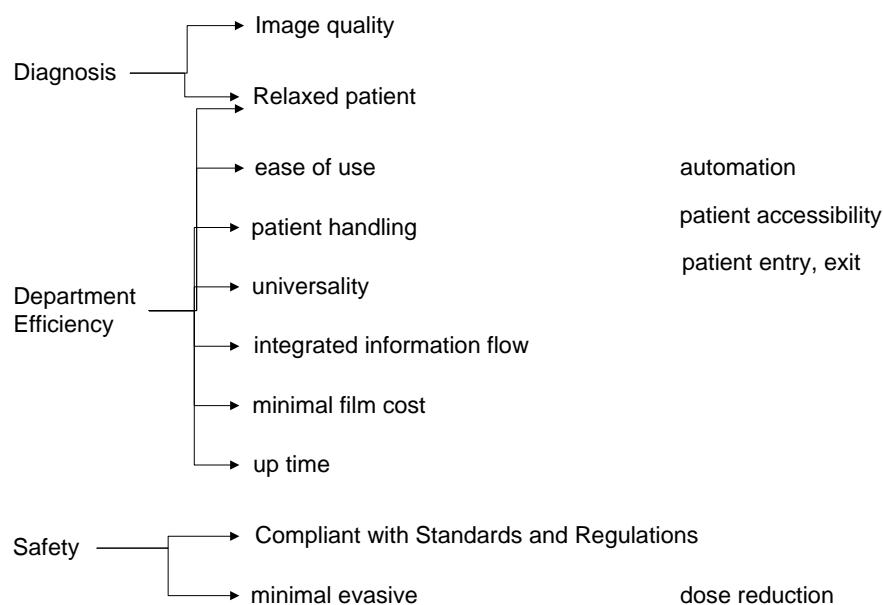


Figure 1.3: Generic drivers of Radiology Departments

Philips Medical Systems core is the imaging equipment in the examination rooms of the radiology department<sup>2</sup>. The key to useful products is the combined knowledge of application (**what**) and technology (**how**).

<sup>1</sup> A poor name for this collection; The main difference is in the maturity of the modality, where this group exists from relative "young" modalities, 20 a 30 years old.

<sup>2</sup> equally important core for Philips Medical Systems is the cardio imaging equipment in the catheterization rooms of the cardiology department, which is out of the Medical Imaging Workstation scope.

## 1.2.2 Historic Phases

The development model of Medical Imaging has changed several times. Roughly the phases in Figure 1.4 can be observed. The first phase can best be characterized as technology development, with poor Market and Application feedback. The next phase overcompensates this poor feedback by focusing entirely on a product.

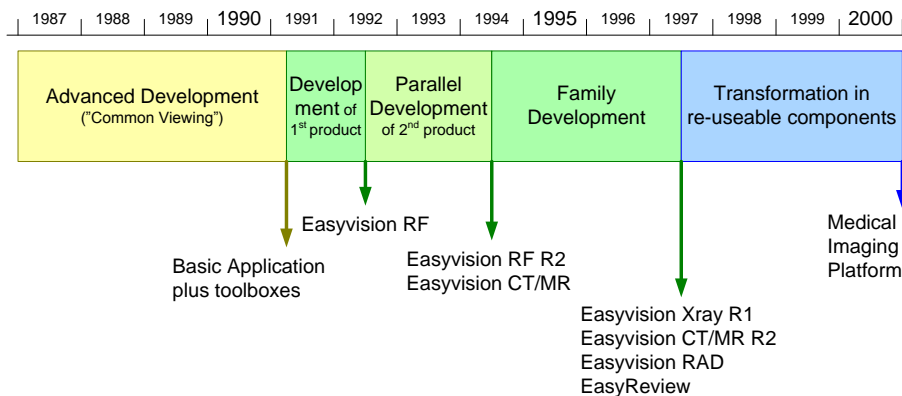


Figure 1.4: Phases of Medical Imaging

Philips Medical Systems has been striving for re-useable viewing components at least from the late seventies. This quest is based on the *assumption* that the viewing of all Medical Imaging Products is so similar, that *cost reduction* should be possible when a common implementation is used. The lessons learned during this long struggle have been partially consolidated in [13].

The group of people, which started the Common Viewing development, applied a massive amount of technology innovations, see Figure 1.5.

## 1.2.3 Basic Application and Toolboxes

The goal of the common viewing development was to create an extensive set of toolboxes, to be used for viewing in all imaging products. The developers of the final products had fine-grain access to all toolboxes. This approach is very flexible and powerful, however the penalty of this flexibility is that the integration is entirely the burden of the product developer.

The power of the toolboxes was demonstrated in a **Basic Application**. This basic application was a superset of all available features and functions. From clinical point of view a senseless product, however a good vehicle to integrate and to demonstrate.

Figure 1.6 shows the idealized layering of the toolboxes and the the Basic Application in september 1991. the toolbox layer builds upon the Sun computing



- standard UNIX based workstation
- full SW implementation, more flexible
- object oriented design and implementation (Objective-C)
- graphical User Interface, with windows, mouse et cetera
- call back scheduling, fine-grained notification
- data base engine, fast, reliable and robust
- extensive set of toolboxes
- property based configuration
- multiple co-ordinate spaces

Figure 1.5: Technology innovations by Common Viewing

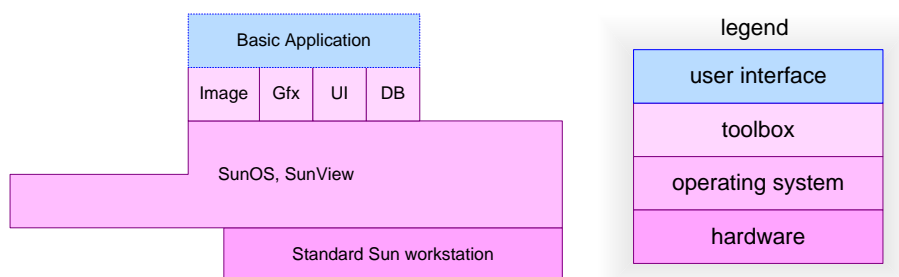


Figure 1.6: Idealized layering of SW toolboxes and Basic Application in september 1991

platform (Workstation, the Sun version of UNIX SunOS and the Sun windowing environment Sunview). The core of common viewing is the imaging and graphics toolbox, and the UI gadgets and style.

#### 1.2.4 Medical Imaging X-Ray

Figure 1.7 shows the X-ray rooms which are involved from the examination until the reading by the radiologist. Around 1990 the X-ray system controls were mostly in the control room, where the operator of the system performed all settings from acquisition setting to printing settings. Some crucial settings can be performed in the room itself, dependent on the application. The hardcopies were produced as literal copies of the screen of the monitor. The printer was positioned at some non-obtrusive place.

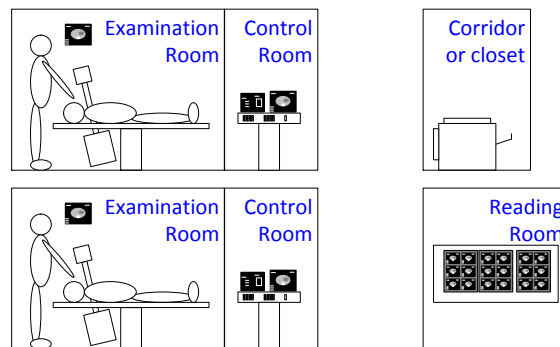


Figure 1.7: X-ray rooms from examination to reading around 1990

The consequence of the literal screen copy was that a lot of redundant information is present on the film, such as patient name, birth date and acquisition settings. On top of that the field of view was supposed to be square or circular, although the actual field of view is often smaller due to the shutters applied.

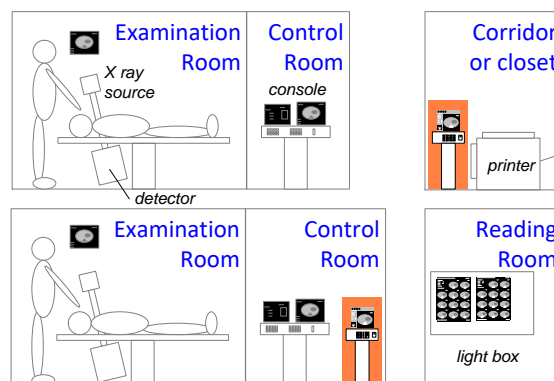


Figure 1.8: X-ray rooms from examination to reading, when Medical Imaging is applied as printserver

The economic existence of Medical Imaging X-ray was based in 1992 on improvements of this printing process. The patient, examination and acquisition information is orderly shown in one viewport, removing all the redundant information near the images itself. A further optimization is applied by a *fit-to-shutter* formatting. These 2 steps together reduce the film use by 20% to 50%.

The user actions needed for the printing are reduced as well, by providing print protocols, which perform the repetitive activities of the printing process. The effectiveness of this automation depends strongly on the application, some applications require quite some fine-tuning of the contrast-brightness, or an essential selection

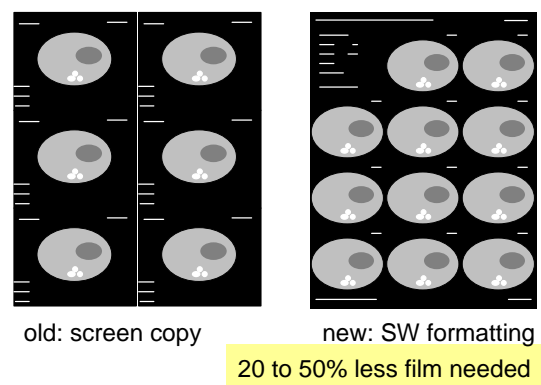


Figure 1.9: Comparison of conventional *screen copy* based film and a film produced by Medical Imaging. This case is very favorable for the Medical Imaging approach, typical gain is 20% to 50%.

step, which require (human) clinical knowhow.

A prominent sales feature at conferences was the 9-button remote control. The elementary viewing functions, such as patient/examination selection, next/previous image and contrast/brightness. This remote control lowered the threshold for clinical personnel, both radiologist as well as technical, enough to catch their interest: The Medical Imaging was not sold as a disgusting computer or workstations, rather it was positioned as a clinical appliance.

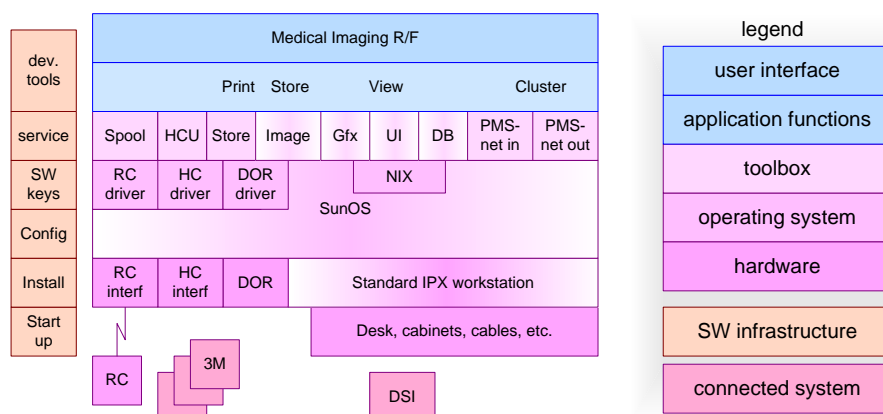


Figure 1.10: Idealized layers of the Medical Imaging R/F software in september 1992

The definition of the Medical Imaging was done by marketing, which described that job as a luxury problem. Normally heavy negotiations were required to get

features in, while this time most of the time marketing wanted to reduce the (viewing and user interface) feature set, in order to simplify the product.

From software point of view the change from basic application to clinical product was tremendous. The grey areas in figure 1.10 indicate new SW. The amount of code increased from 100 klines to 350 klines of code.

### 1.2.5 Second Concurrent Product: Medical Imaging CT/MR

Up to 1992 the Medical Imaging organization had a single focus, first on toolboxes, later on Medical Imaging R/F. In 1993 it was decided to apply the Medical Imaging also on CT and MR.

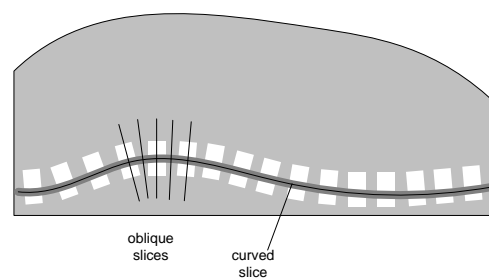


Figure 1.11: Example of Multi Planar Reformating applied on the spine

The printing functionality of CT and MR scanners improves significantly when Medical Imaging is applied as printserver. However the CT and MR applications can benefit also from interactive functionality, more than the X-ray applications. An clear example is the Multi Planar Reformating (MPR) functionality, where arbitrary slices are reconstructed from the volume data set.

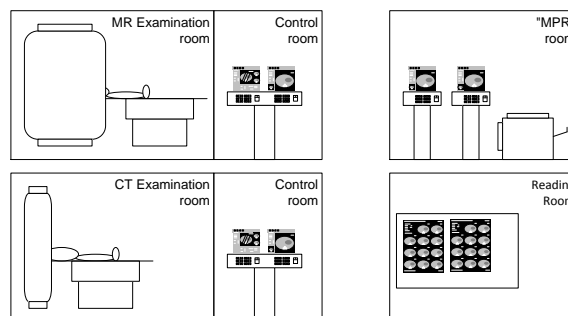


Figure 1.12: Example of CT and MR department, where Medical Imaging is deployed

Superficially X-ray viewing looks the same as CT and MR viewing. However

the viewing is different in many subtle ways. A fundamental difference is that X-ray images are *projection* images, while CT and MR images are *slices*, which means that CT and MR images have a 3D "meaning", which is missing in X-ray images. The 3D relationship is amongst others used for navigation, a *point-and-click* type of user interface: clicking on a scanogram immediately shows the related slice(s) at that position.

The software was significantly extended, the code size increased from 350 klines to 600 klines. Note that this is not only an extension with 250 klines, from the original 350 klines roughly half was modified or removed. In other words a significant amount of refactoring has taken place concurrent with the application extensions. Figure 1.13 shows the (idealized) SW structure at the completion of Medical Imaging CT/MR and the second release of Medical Imaging R/F. Light grey blocks represent new code, dark grey represents major redesigns.

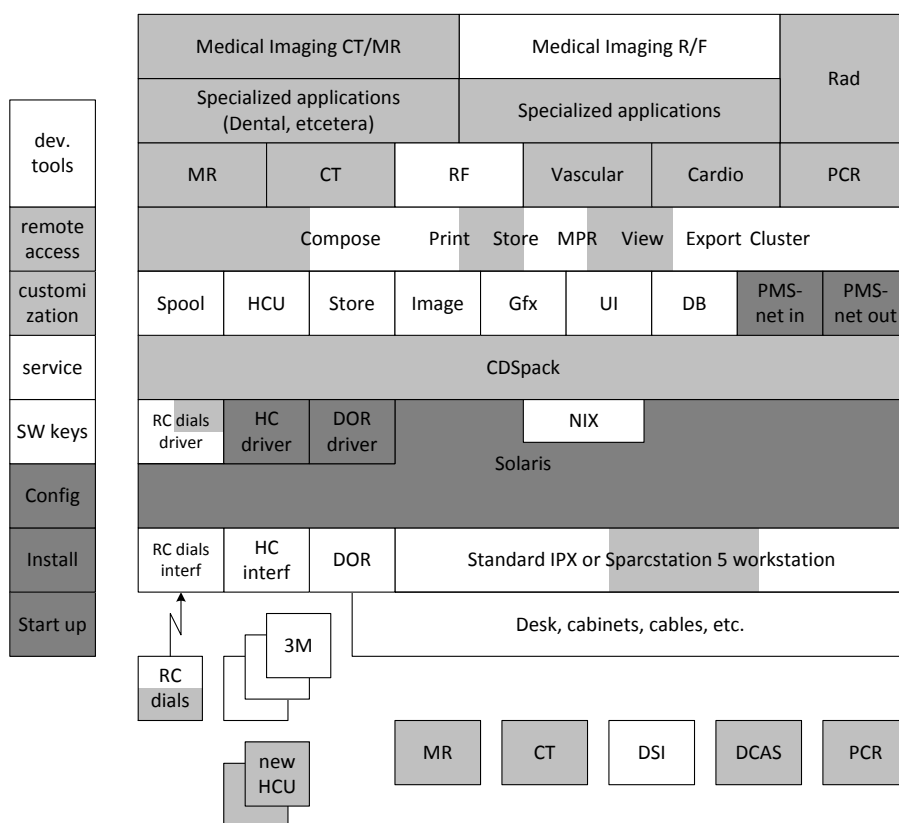


Figure 1.13: Idealized layers of the Medical Imaging software in June 1994

All diagrams 1.6, 1.10 and 1.13 are labelled as *idealized*. This adjective is used because the actual software structure was less *well structured* than presented by

these diagrams. Part of the refactoring in the 1992-1994 time frame was a cleanup, to obtain well defined dependencies between the software-”groups”. These groups were more fine-grained than the blocks in these diagrams.

### 1.2.6 Towards Workflow

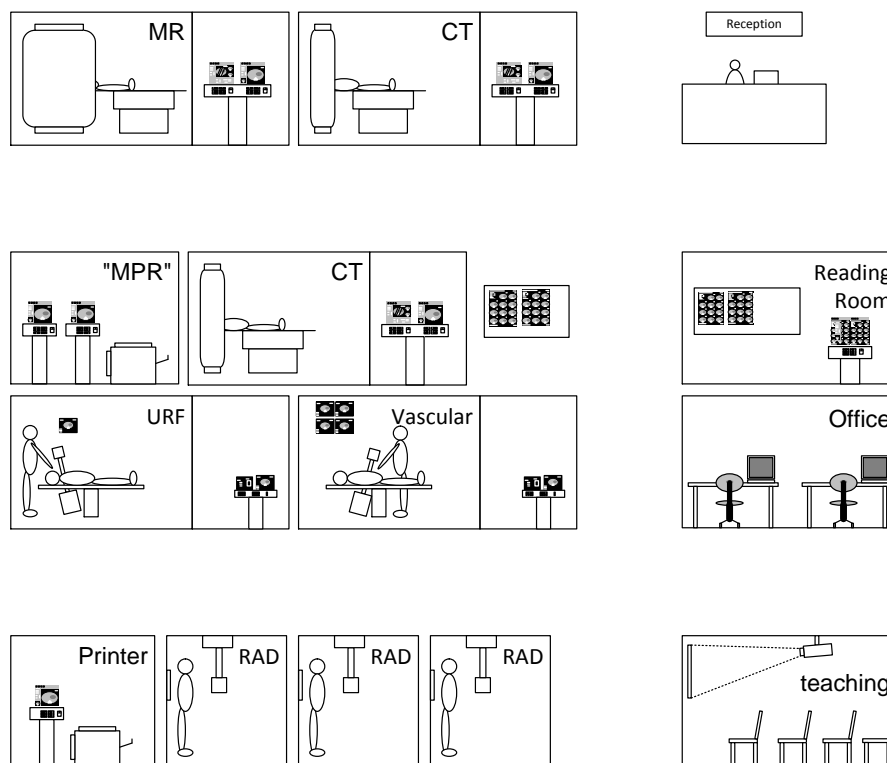


Figure 1.14: Radiology department as envisioned in 1996

Medical Imaging R/F and Medical Imaging CT/MR were positioned as *modality enhancers*. The use of these systems enhances the value of the modality. They are used in the immediate neighborhood of the modality, before the reporting is done. From sales point of view these Medical Imaging are additional options for a modality sales.

The radiology workflow is much more than the acquisition of the images. Digitalization of the health-care information flow requires products which fit in the broader context of radiology and even the diagnostic workflow. Figures 1.14 and 1.15 show the increasing context where the workstation technology can be deployed.

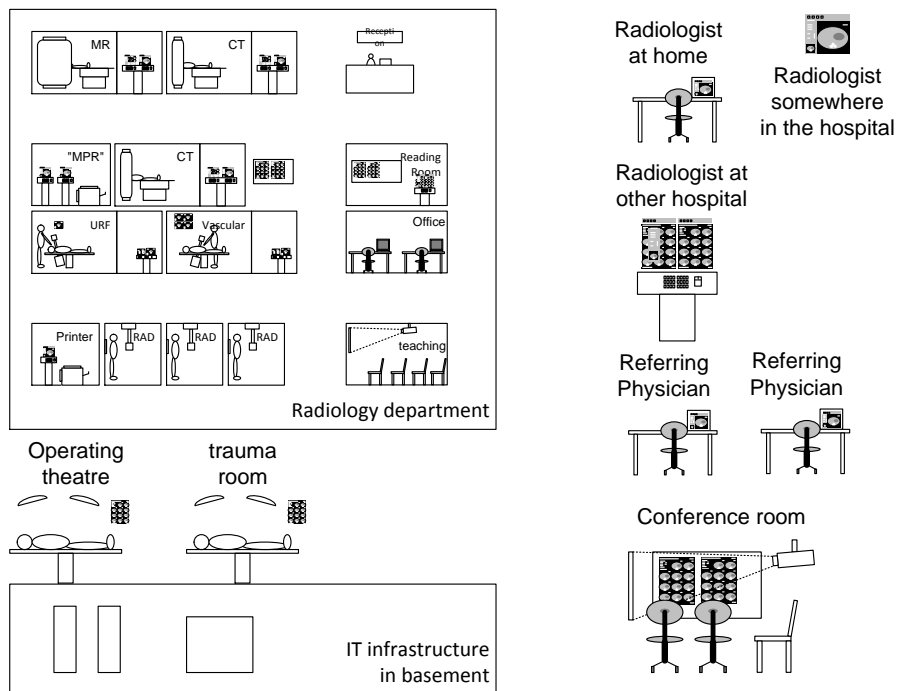


Figure 1.15: Medical Imaging in health-care workflow perspective, as envisioned in 1996

The increasing context causes new extensions of the SW building, as shown in Figure 1.16.

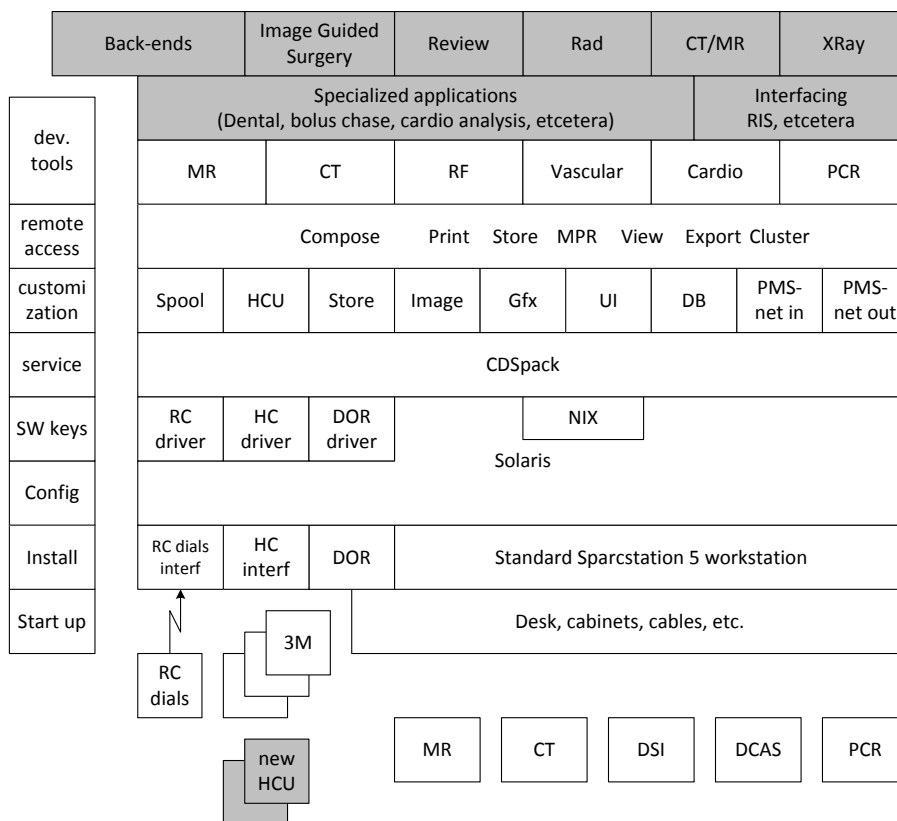


Figure 1.16: Idealized layers of the Medical Imaging software in 1996



## 1.3 Architecture

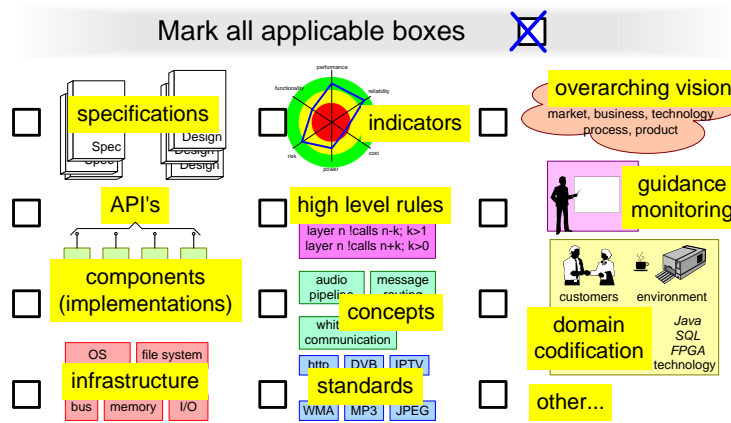


Figure 1.17: What is Architecture?

What is *Architecture*? Every individual appears to use their own definition of architecture. Figure 1.17 shows many different aspects that are frequently mentioned as being part of the architecture.

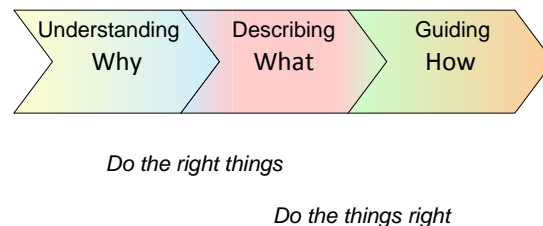


Figure 1.18: What is Architecture?

We will use a broad definition of *Architecture*. Architecture is the combination of the know how of the solution (technology) and understanding of the problem (customer/application). The architect must play an independent role in considering all stakeholders interests and searching for an effective solution. The fundamental architecting activities are depicted in figure 6.1.

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 designers and engineers. Guiding the implementation is done by providing guidelines and high level designs for many different viewpoints. Figure 6.16 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 6.16

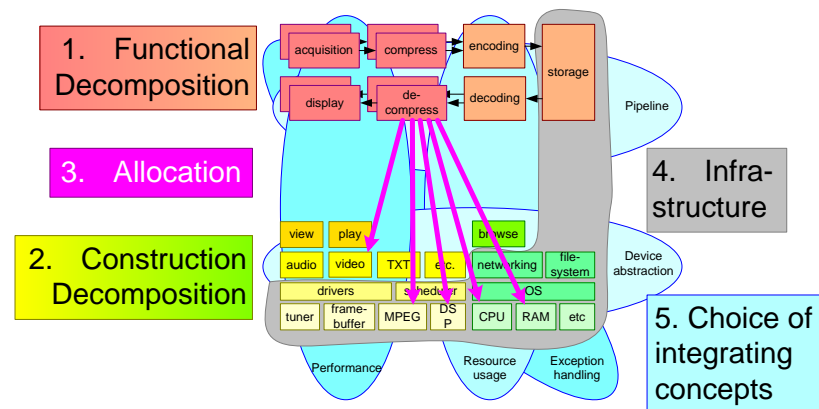


Figure 1.19: "Guiding How" by providing rules for:

shows that architecting involves many more aspects and especially the integrating concepts are crucial to get working products.

Architecting involves amongst others *analyzing, assessing, balancing, making trade-offs* and *taking decisions*. This is based on architecture information and **facts**, following the needs and addressing the **expectations** of the stakeholders. A lot of the architecting is performed by the architect, which is frequently using **intuition**. As part of the architecting *vision, overview, insight* and *understanding* are created and used.

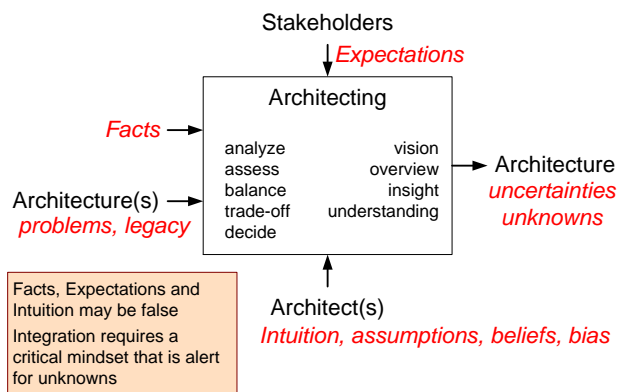


Figure 1.20: The Art of Architecting

The strength of a good architect is to do this job in the real world situation, where the **facts**, **expectations** and intuition sometimes turn out to be false or changed! Figure 6.17 visualizes this art of architecting.

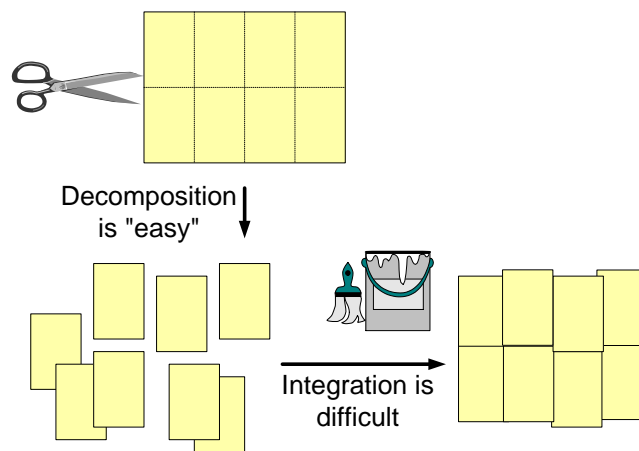


Figure 1.21: Architecting is much more than Decomposition

Many people expect the architect to decompose, as mentioned in the explanation of "guiding how", while integration is severely underestimated, see figure 6.18. 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.

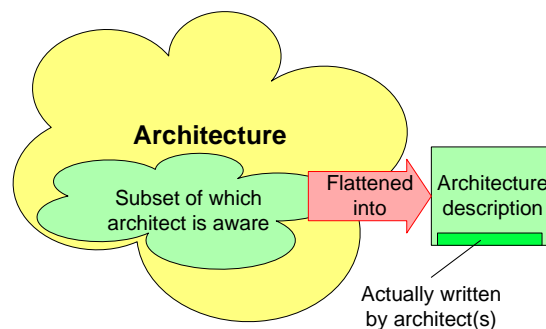


Figure 1.22: The architecture description is by definition a flattened and poor representation of an actual architecture.

IEEE 1471 makes another interesting step: it discusses the *architecture description* not the *architecture* itself. The *architecture* is used here for the way the system is experienced and perceived by the stakeholders<sup>3</sup>.

This separation of *architecture* and *architecture description* provides an interesting insight. The *architecture* is infinite, rich and intangible, denoted by a cloud

<sup>3</sup>Long philosophical discussions can be held about the definition of **the** architecture. These discussions tend to be more entertaining than effective. Many definitions and discussions about the definition can be found, for instance in [7], [5], or [9]

in figure 1.22. The *architecture description*, on the other hand, is the projection, and the extraction of this rich *architecture* into a flattened, poor, but tangible description. Such a description is highly useful to communicate, discuss, decide, verify, et cetera. We should, however, always keep in mind that the description is only a poor approximation of the *architecture* itself.

## 1.4 Platform

Many people advocate generic developments, such as platforms, claiming a wide range of advantages. Effective implementation of generic development has proven to be quite difficult. Many attempts to achieve these claims by generic developments have resulted in the opposite goals, such as increased time to market, quality and reliability problems et cetera. We need a better rationale to do generic developments, in order to design an effective platform creation process.

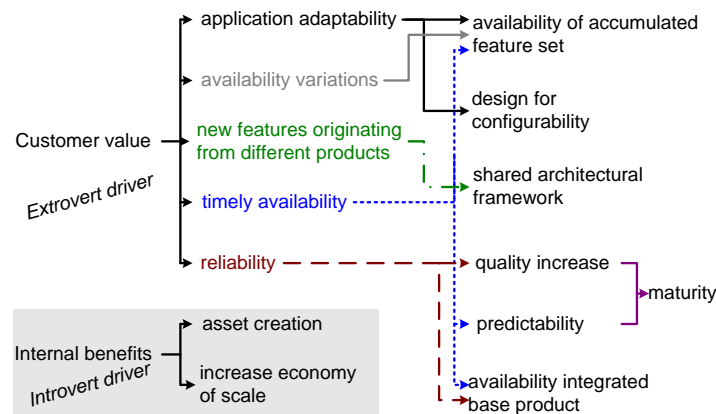


Figure 1.23: Drivers of Generic Developments

Figure 1.23 shows drivers for Generic Developments and the derived requirements for the Generic Something Creation Process. The first driver (*Customer value*) is extrovert: does the product have value for the customer and is he willing to buy the product? The second driver *Internal Benefits* is introvert, it is the normal economic constraint for a company.

Today high tech companies are knowhow and skill constrained, in a market which is extremely fast changing and which is rather turbulent. Cost considerations are degraded to an economic constraint, which is orders of magnitude less important than being capable to have valuable and sellable products.

The derivation of the requirements shows clearly that these requirements are not a goal in itself. For instance an shared architecture framework is required to enable features developed for one product to be used in other products as well, which in turn should have value for a customer. So the verification of this requirement is to propagate a new valuable feature from one product to the next, with small effort and lead time.

These drivers and requirements derivation is emphasized, because many generic developments result in large monolithic general purpose things, fulfilling:

- availability accumulated feature set

- designed for configurability
- shared architectural framework
- mature

without bringing any customer value; "You can not have this easy shortcut, because our architectural framework does not support it, changing the framework will cost us 100 man-years in 3 years elapsed time"

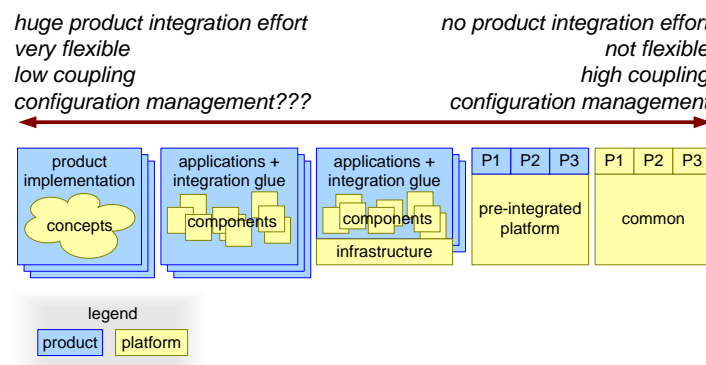


Figure 1.24: What is a Platform?

But what is a *platform*? Many different types of platforms can be found. Figure 1.24 shows a classification of platforms along an axis of increasing content and integration. The “lightest” platform is a shared set of concepts, where every product implements its own instantiation. The most “heavy” platform is the implementation of a superset of all products, where the creation of a product *only* involves a configuration step of selecting the right functionality and performance. The figure shows some intermediate possibilities, from light to more heavy respectively: a collection of shared implementations of components, the same plus infrastructure, and a complete pre-integrated framework. *Light* platforms require lots of integration effort, are very flexible, have low coupling, and require a lot of complex configuration management effort. *Heavy* platforms do not require much integration, are not flexible, create lots of coupling between products, and require less complex configuration management at the expense of coupled release cycles.

The platform development results in deliverables. To support integration and trouble shooting the delivery of source information is recommended. Black box reuse tends to create surprises, due to invisible consequences. However, delivering the source code itself, creates additional requirements. The source code is only useful if the *development environment*, *specifications*, *configuration management*, *documentation tools*, *development process* and guidelines for the *infrastructure* are also provided. Figure 1.25 shows these deliverables, and Figure 1.26 shows the

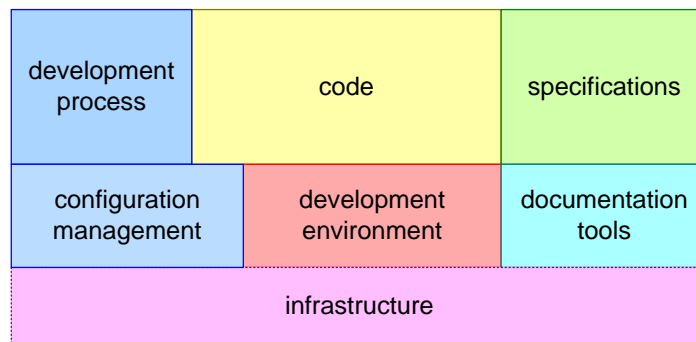


Figure 1.25: Platform Source Deliverables

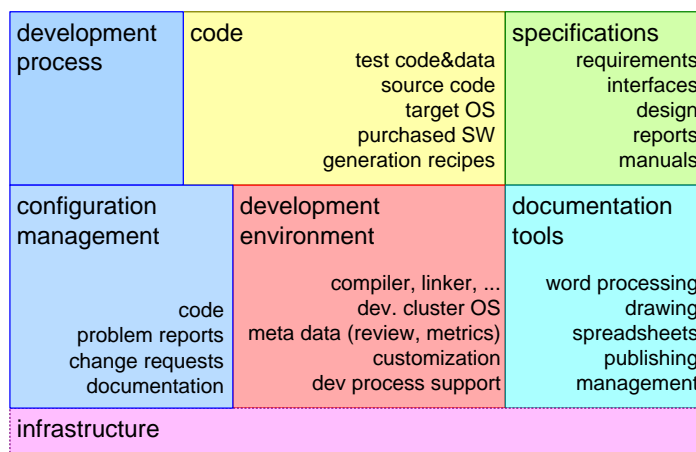


Figure 1.26: And now in More Detail...

same deliverables with more detailed content. The message of this last figure is that much more is involved in platform development than a set of source code files.

The case, as shown in Section 1.2, used a platform approach to share common functions. In the table in Figure 1.27 the efficiency of this platform approach is evaluated. The basis for this evaluation is the number of different applications that has been realized and the required effort. This table shows that 13 persons were needed per application in 1993, while in 1996 only 3 persons per application were needed. The re-use of lower level functions facilitated a more efficient application development process. In practice the lead-time reduction of new applications was even more important. A rich and flexible platform is also a rapid prototyping vehicle. This last argument is far from trivial: many platforms are large and complex and do not facilitate rapid prototyping at all!

		1992	1993	1994	1995	1996
<i>value metric</i>	applications	1	4	8	16	32
	number of inputs (a.o. modalities)	1	5	10	15	
<i>number of people</i>	platform			35	37	38
	applications			27	35	41
	total		52	62	72	79
<i>efficiency</i>	people per application		13	8	5	3

Figure 1.27: Example of Platform Efficiency

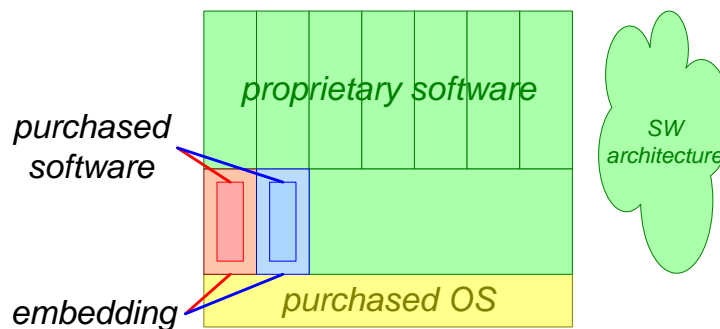


Figure 1.28: Purchased SW Requires Embedding

A complicating factor is the use of COTS (Commercial Of The Shelf) software. Software developed as part of a platform follows the architecture guidelines of the platform. However, purchased software has been developed independent of the platform, using it's own architecture guidelines. Figure 1.28 shows that purchased software requires some kind of embedding to fit it into the desired architecture.

Figure 1.29 zooms in on the typical additional efforts to embed purchased software in a platform. Most embedding effort is required to ensure the desired system level behavior and qualities: configuration, installation, start-up and shutdown et cetera.

The mismatch of existing platform software and purchased software results in lots of unwanted side-effects. Figure 5.13 shows a number of these unwanted side-effects. The side-effects cause the addition 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.



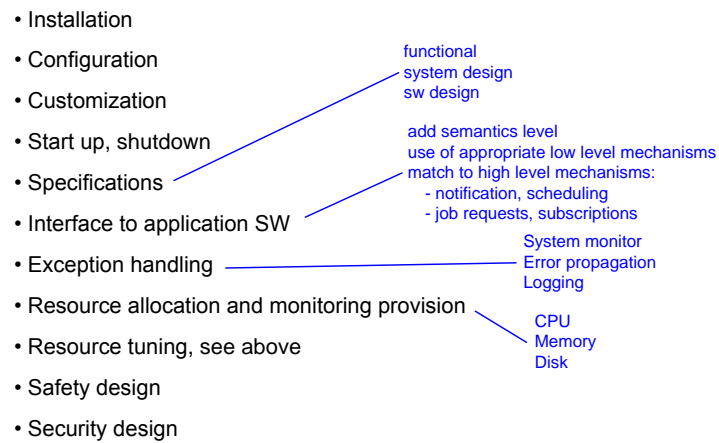


Figure 1.29: Embedding Costs of Purchased SW

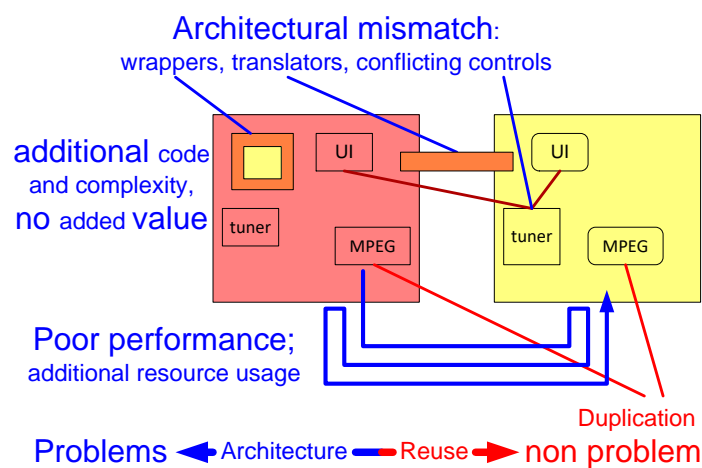


Figure 1.30: Example of Embedding Problems

## 1.5 The Time Dimension

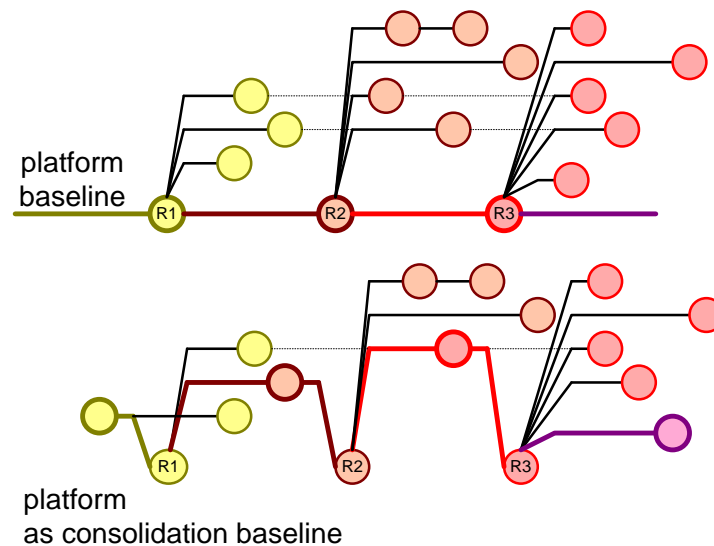


Figure 1.31: Who is First: Platform or Product?

Many philosophies are practiced to synchronize platforms and products. The main choice is the primary vehicle for change:

- innovate in products and consolidate in a platform
- innovate in the platform and propagate to products

These two variants are visualized in Figure 1.31.

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 6.19 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 served a dynamic fast changing market. In other words it is a miracle if a platform is stable, when both the supplying as well as the consuming side are not stable at all.

The more academical oriented methods propose a "first time right approach". This sounds plausible, why waste time on wrong implementations first? The practical problem with this type of approach is that it does only work in very specific circumstances:

- well defined problem
- few people (few background, few misunderstandings)

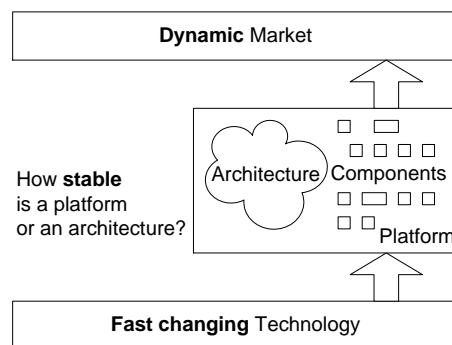


Figure 1.32: Myth: Platforms are Stable

- appropriate skill set (the so-called "100%" instead of "80/20" oriented people)

The first clause for our type of products is nearly always false, remember the dynamic market. The second clause is in practical cases not met (100+ manyear projects), although it might be validly pointed out that the size of the projects is the cause of many problems. The third clause is very difficult to meet, I do know only a handful of people fitting this category, none of them making out type of products (for instance professors).

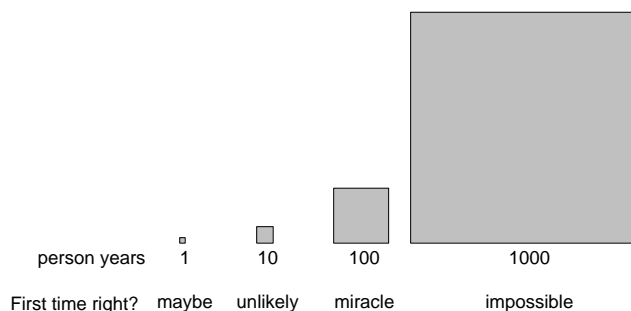


Figure 1.33: The first time right?

Figure 6.20 shows the relationship between team size and the chance of successfully following the *first time right* approach.

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

The evolution of a platform is illustrated in figure 6.22 by showing the change in the Easyvision [16] 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

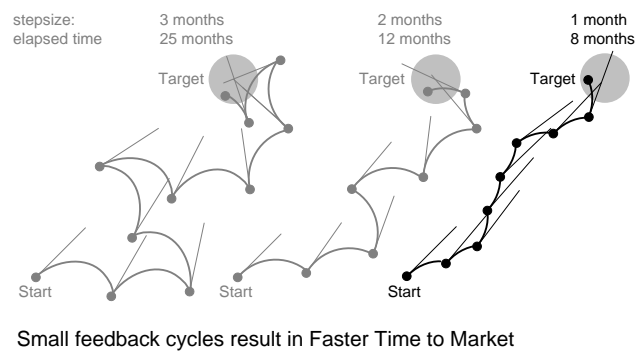


Figure 1.34: Feedback (3)

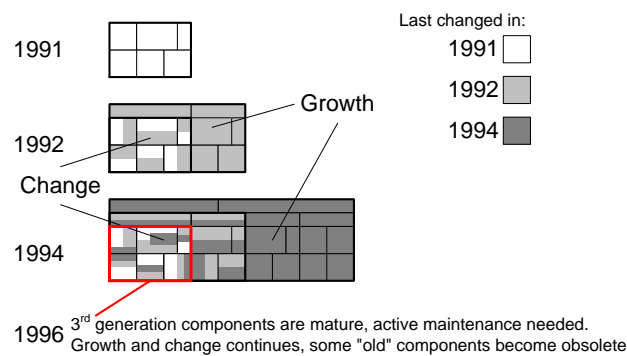


Figure 1.35: Platform Evolution (Easyvision 1991-1996)

existing code base is touched by changes.

The business context, the application, the product and its components have all their own specific life-cycles. In Figure 1.36 several different life-cycles are shown. The application and business context in the customer world are shown at the top of the figure, and at the bottom the technology life-cycles are shown. Note that the time-axis is exponential; the life-cycles range from one month to more than ten years! Note also the tension between commodity software and hardware life-cycles and software release life-cycles: How to cope with fast changing commodities? And how to cope with long living products, such as MR scanners, that use commodity technologies?

Figure 1.37 shows a reference model for image handling functions. This reference model is classifying application areas on the basis of those characteristics that have a great impact on design decisions, such as the degree of distribution, the degree and the cause of variation and life-cycle. Such a reference model is one of the means to cope with widely different life-cycles.

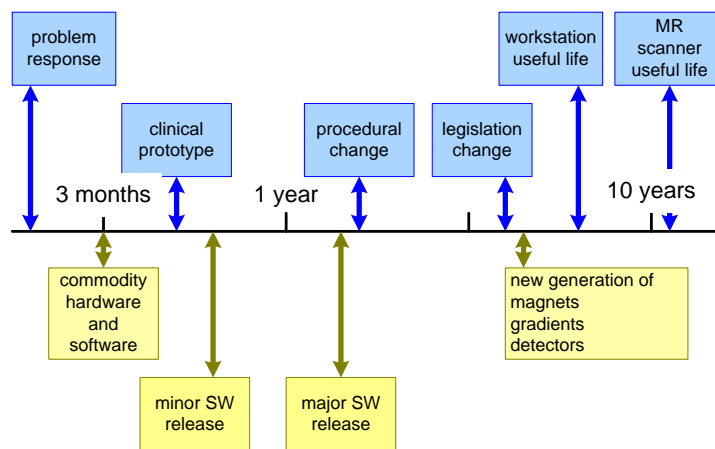


Figure 1.36: Life-cycle Differences

*Imaging and treatment* functions are provided of modality systems with the focus on the patient. Safety plays an important role, in view of all kinds of hazards such as radiation, RF power, mechanical movements et cetera. The variation between systems is mostly determined by:

- the acquisition technology and its underlying physics principles.
- the anatomy to be imaged
- the pathology to be imaged

The complexity of these systems is mostly in the combination of many technologies at state-of-the-art level.

*Image handling* functions (where the medical imaging workstation belongs) are distributed over the hospital, with work-spots where needed. The safety related hazards are much more indirect (identification, left-right exchange). The variation is more or less the same as the modality systems: acquisition physics, anatomy and pathology.

The *information handling* systems are entirely distributed, information needs to be accessible from everywhere. A wide variation in functionality is caused by “social-geographic” factors:

- psycho-social factors
- political factors
- cultural factors
- language factors

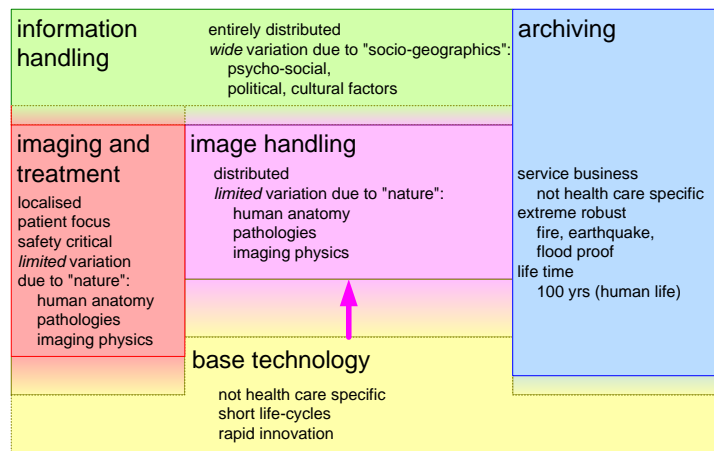


Figure 1.37: Reference model for health care automation

These factors influence what information must be stored (liability), or must not be stored (privacy), how information is to be presented and exchanged, who may access that information, et cetera.

The *archiving* of images and information in a robust and reliable way is a highly specialized activity. The storage of information in such a way that it survives fires, floods, and earthquakes is not trivial<sup>4</sup>. Specialized service providers offer this kind of storage, where the service is location-independent thanks to the high-bandwidth networks.

All of these application functions build on top of readily available IT components: the *base technology*. These IT components are innovated rapidly, resulting in short component life-cycles. Economic pressure from other domains stimulate the rapid innovation of these technologies. The amount of domain-specific technology that has to be developed is decreasing, and is replaced by base technology.

<sup>4</sup>Today terrorist attacks need to be included in this list full of disasters, and secure needs to be added to the required qualities.

## 1.6 Process View

The business process for an organization that creates and builds systems consisting of hardware and software is decomposed in four main processes as shown in figure 1.38.

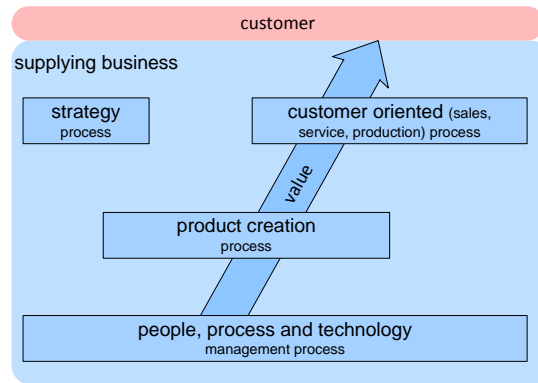


Figure 1.38: Simplified decomposition of the business in 4 main processes

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 cash-flow 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 cash-flow 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.

**Strategy 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.

The simplified process description given in figure 1.38 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 3.25 shows the modified process decomposition

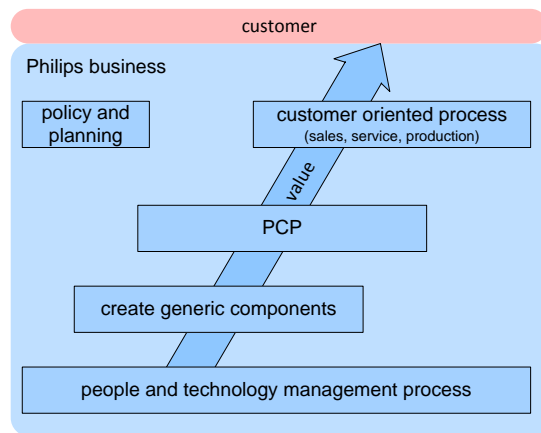


Figure 1.39: Modified Process Decomposition

(still simplified of course) including this additional process "Generic Something Creation Process".

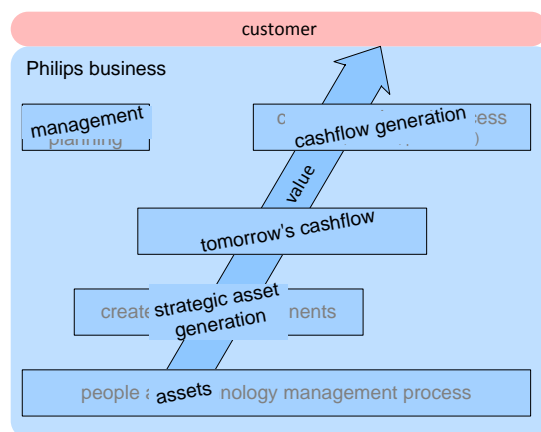


Figure 1.40: Financial Viewpoint on Process Decomposition

Figure 3.26 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 cash-flow.

The consequence of this additional process is an lengthening of the value chain and consequently a longer feedback chain as well. This is shown in figure 3.27. The increased length of the feedback chain is a significant threat for generic 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



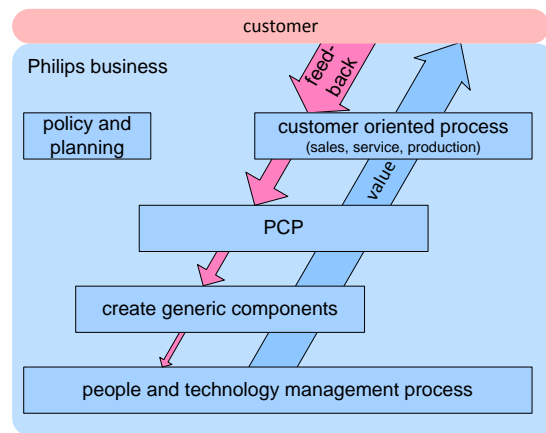


Figure 1.41: Feedback flow: loss of customer understanding!

product, which is released to be used by the product developments.

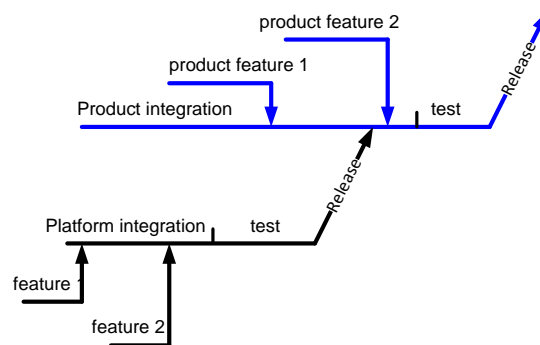


Figure 1.42: The introduction of a new feature as part of a platform causes an additional latency in the introduction to the market.

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 3.29.

The list of pitfalls in Figure 1.43 has been compiled on the basis of many disastrous or halfway successful efforts of platform 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



Figure 1.43: Sources of failure in platform developments

shown in figure 1.44.

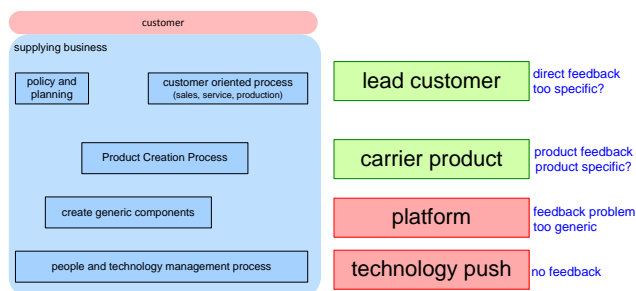


Figure 1.44: Models for SW reuse

### 1.6.1 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.

### **1.6.2 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.

### **1.6.3 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.

## 1.7 Market Driven

A useful top level decomposition of an architecture is provided by the so-called “CAFCR” model, as shown in figure 1.45. The *customer objectives* view and the *application* view provide the **why** from the customer. The *functional* view describes the **what** of the product, which includes (despite the name) also the *non functional* requirements. The **how** of the product is described in the *conceptual* and *realization* view, where the conceptual view is changing less in time than the fast changing realization (Moore’s law!).

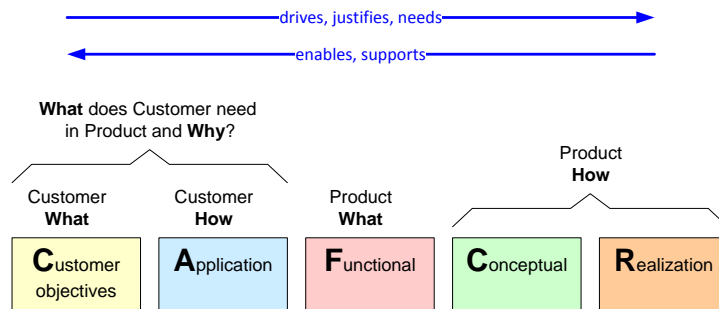


Figure 1.45: The “CAFCR” model

The job of the architect is to integrate these views in a consistent and balanced way. Architects do this job by *frequent viewpoint hopping*, looking at the problem from many different viewpoints, sampling the problem and solution space in order to build up an understanding of the business. Top down (objective driven, based on intention and context understanding) in combination with bottom up (constraint aware, identifying opportunities, know how based), see figure 1.46.

In other words the views must be used concurrently, not top down like the waterfall model. However at the end a consistent story must be available, where the justification and the needs are expressed in the customer side, while the technical solution side enables and support the customer side.

The model will be used to provide a next level of reference models and methods. Although the 5 views are presented here as sharp disjunct views, many subsequent models and methods don’t fit entirely in one single view. This in itself not a problem, the model is a means to build up understanding, it is not a goal in itself.

One of the key success factors of platform development is *scoping*. The opposing forces are the efficiency drive by higher management teams, increasing the scope, and the need for customer specifics by project teams, minimizing the platform scope. Scope overstretching is one of the major platform pitfalls: in best case the result is that the organization is very efficient, but customers are dissatisfied. Worst case the entire organization drowns in the overwhelming complexity. Blindly

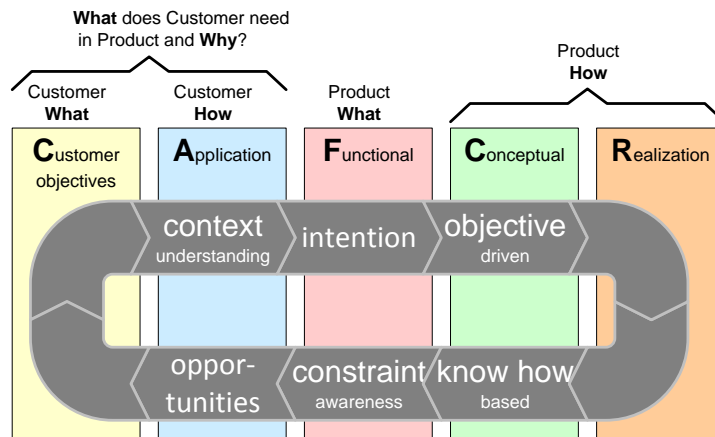


Figure 1.46: Five viewpoints for an architecture. The task of the architect is to integrate all these viewpoints, in order to get a *valuable, usable* and *feasible* product.

following (potential) customers is another pitfall: best case we end up with a satisfied customer and a good starting point for next products. The real challenge is to build up sufficient understanding to find the sweet spot for the platform scope: efficient by leveraging synergy, but sufficiently agile to be responsive to customers.

Figure 2.11 shows an example of platform scoping. In this case the synergy of the producer is in technologies, such as Closed Circuit TV (CCTV), audio, broadcasting, access control, and networking. These technologies are used in widely differing application domains: airports, railway stations, intelligent buildings and motorway management systems. These heterogeneous domains can share a platform, as long as the functionality is restricted to the shared technologies. The analysis

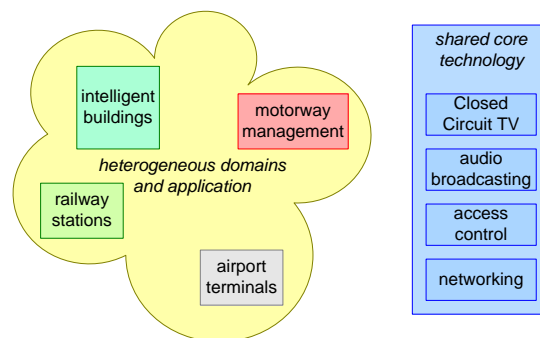
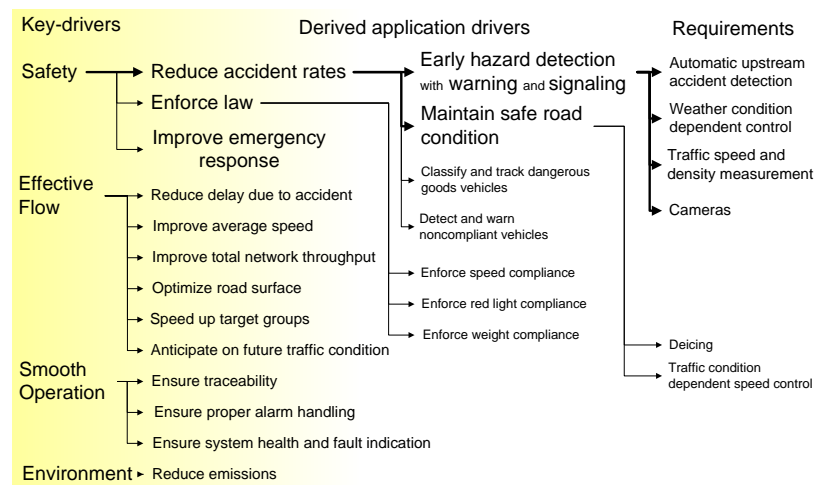


Figure 1.47: Example of Scoping of a Platform.

to find the right level of synergy is based on the key driver method [14]. The essence of the objectives of the customers can be captured in terms of customer key drivers. The key drivers provide direction to capture requirements and to focus the development. The key drivers in the customer objectives view will be linked with requirements and design choices in the other views. The key driver submethod gains its value from relating a few sharp articulated key drivers to a much longer list of requirements. By capturing these relations a much better understanding of customer and product requirements is achieved.



*Note: the graph is only partially elaborated for application drivers and requirements*

Figure 1.48: Example of the four key drivers in a motorway management system

Figure 1.48 shows an example of key drivers for a motorway management system, an analysis performed at Philips Projects in 1999. The same method has been applied on the other domains.

The key drivers and design decisions can be visualized as a *thread of reasoning* [23] during the development of products and platform. This thread of reasoning captures the essential relations between customers needs and technological decisions, with emphasis on tensions and trade-offs. Figure 1.49 shows an example of such a thread of reasoning for the Medical Imaging Workstation.

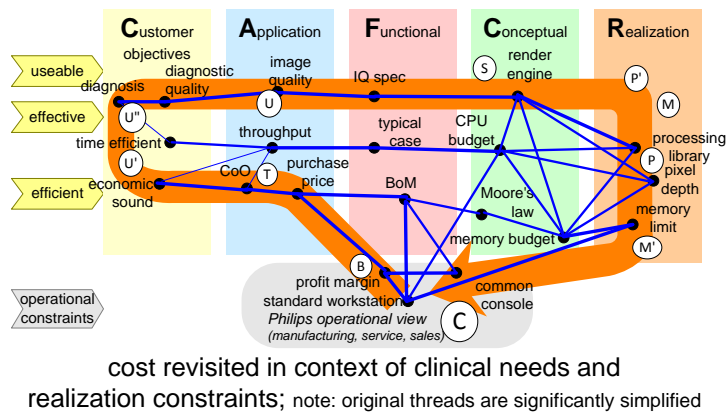


Figure 1.49: Example Thread of Reasoning from the Medical Imaging Workstation

## 1.8 Recommendations

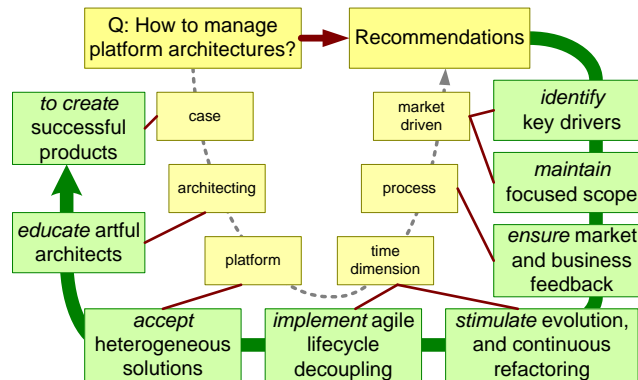


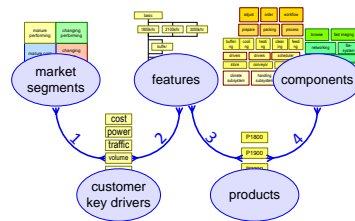
Figure 1.50: Summary of recommendations to manage platform architectures

Figure 1.50 summarizes the recommendations to manage platform architectures. We traverse in the opposite direction of the description in this paper. Identification of the *key drivers* is the first step in understanding the essence from market point of view. The key drivers are used to define the platform *scope*; a well defined scope provides focus to the development organization. The process of developing a platform requires special attention for frequent and to-the-point feedback from the business and the market. The time dimension emphasizes the many different rhythms in product and platform development and the dynamics of both application and technology. The recommendation to cope with rhythms and dynamics is to stimulate evolutionary approaches and to invest sufficiently in continuous refactoring of the architecture. Also *agile* life-cycle decoupling facilitates the different rhythms and dynamics. For the platform itself it is important to understand the versatility and the heterogeneity involved. Platform development should avoid dogmatic unification, instead recognition of heterogeneous solution results in more robust platform development. The overall activities described so far require a few skilled and artful architects. Artful means creative, open minded, humans; the complexity and dynamics of the context does not allow for mechanistic or dogmatic solutions. Satisfying all of the recommendations will help to create nice, innovative and successful products!



## Chapter 2

# A Method to Explore Synergy between Products



### 2.1 Introduction

	Customer objectives	Application	Functional	Conceptual	Realization
<b>Multiple markets</b>	different customers	different applications	similar products	shared concepts	shared technology
for example electron microscope markets:	material sciences life sciences semiconductors	EM specialists biologists process quality	everything possible specific handling high throughput	e-beam sources, optics vacuum acquisition control	
<b>Single market</b>	same customers	different applications & stakeholders	different products	shared concepts	shared technology
for example, health care, radiology market	radiology department	gastrointestinal orthopedics neurology	radiography x-ray diagnostics MRI, CT scanner viewing	patient support patient information image information storage & communication	

Figure 2.1: Types of synergy

We can distinguish two types of situations where we can strive to harvest synergy, as illustrated in Figure 2.1:

**Single market, different products** where the customer world is homogeneous, while products can be quite heterogeneous in both concepts and technologies that are used.

**Multiple markets, quite similar products** where the customer world is heterogeneous, while the products are different, but quite similar. The similarity in the products suggests that synergy is present that can be harvested.

Figure 2.1 also shows one example in both categories. The radiology department in health care is an example of a homogeneous market, where many different products are interoperating to provide the desired capabilities. Some of these systems are diagnostic equipment with different imaging modalities, e.g. X-ray systems, Magnetic Resonance Imaging, Computer Tomography. However, also information technology systems are used for administration, viewing, communication, and archiving. Some functionality is quite similar between these different systems, and hence might result in synergy opportunities.

## 2.2 Stepwise method to explore synergy opportunities

explore markets, customers, products and technologies	
share market and customer insights	
identify product features and technology components	
make maps:	market segments - customer key drivers customer key drivers - features features - products products - components
discuss value, synergy, and (potential) conflicts	
create long-term and short-term plan	

Figure 2.2: Approach to Platform Business Analysis

Figure 2.2 shows the stepwise method to explore and analyze opportunities to harvest synergy.

**Explore markets, customers, products and technologies** to create a shared understanding of the playing field.

**Share market and customer insights** by studying one customer and one product, followed by a more extensive study of work flows.

**Identify product features and technology components** by doing initial specification and design work.

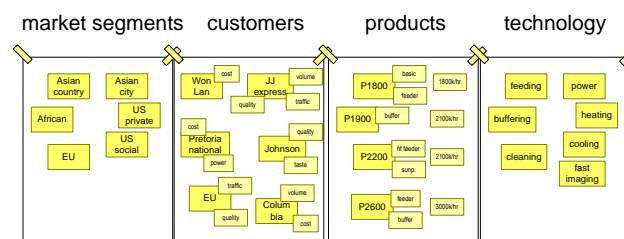
**Make maps** where the views that resulted from the first steps are related.

**Discuss value, synergy and (potential) conflicts** to get the main issues on the table in a factual way.

**Create long term and short term plan** to transform what can be done into something that (probably) will be done.

The whole process described by this method should be performed by an exploration team, a small team of key people, including marketing managers, architects, and key technology experts.

## 2.2.1 Explore markets, customers, products and technologies



*brain storm and discuss time-boxed*

Figure 2.3: Explore Markets, Customers, Products and Technologies

The exploration is performed by using fixed time boxes to discuss the following questions by the exploration team:

- What markets do we want to serve?
- What specific customers do we expect? What are the key concerns per customer?
- What products do we foresee? What are key characteristics of these products?
- What technologies do we need?

The purpose is to make a quick scan of the playing field so that a shared insight is created between the members of the team. Figure 2.3 shows the typical result of the exploration: a number of flip-charts with sticky notes. This first scan can be done in a half day to a full day.

## 2.2.2 Share market and customer insights

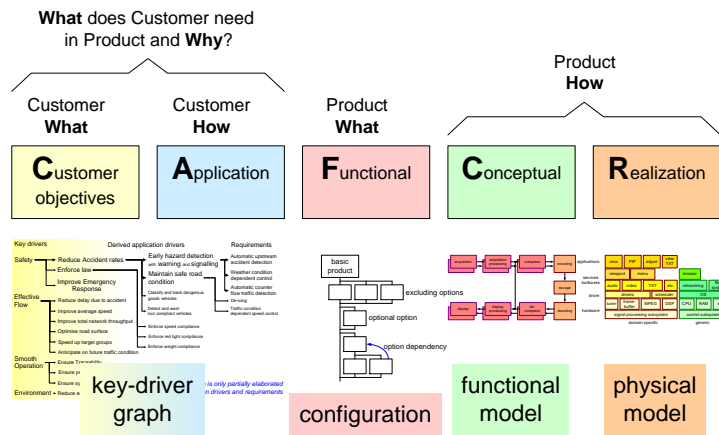


Figure 2.4: Study one Customer and Product

The challenge is to get more substance after the first quick scan. Figure 2.4 shows how the CAFCR model is followed to explore one product for one market in more depth. The idea is that one such depth probe helps the team to get a deeper understanding that can be extended to other product by variations on an theme. In this figure the CAF-views are covered by a key driver graph, the F-view focuses on the required commercial product structure, The Conceptual view is used for a functional model of the system internals and the R view shows a block diagram. This is an example of a CAFCR analysis, but specific markets and products can benefit from other submethods in the CAFCR views.

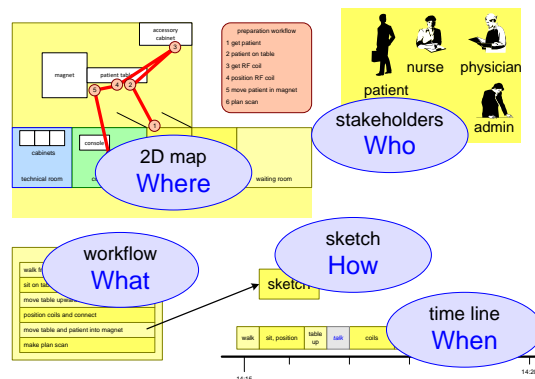


Figure 2.5: Work Flow Analysis for Different Customers/Applications

The next step in digging in deeper is to explore the work flow of different

customers. Figure 2.5 shows the different perspectives on the customer work flow:

**Where** are work flow steps performed?

**What** is done in the work flow?

**Who** is involved

**When** are steps performed, and what is the duration?

**How** are selected steps performed?

A specific insight in the work flow of different customers and applications is critical for later choices about synergy. This step is too often skipped, either because of time pressure or because of ignorance. Insufficient understanding of the use compromises the products and hence degrades the value for classes of customers.

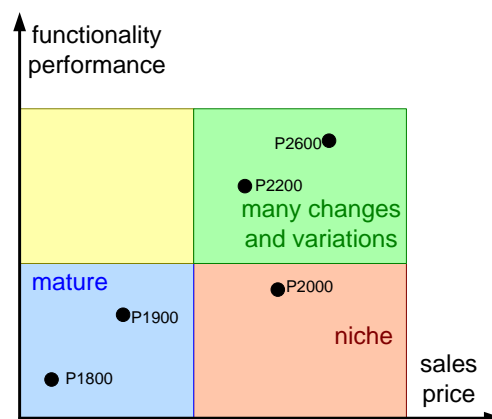


Figure 2.6: Make Map of Customers and Market Segments

At this moment the exploration team has insight in different customers. It helps the team and its stakeholders if the growing insight of these different customers and their needs for products can somehow be captured in a single map with a few main characteristics. Figure 2.6 shows a simplistic example. Often characteristics such as price and performance parameters are used for such map.

### 2.2.3 Identify product features and technology components

In this step the commercial structure of the product is further elaborated: What are the required commercial configurations, what should be optional? Also the construction decomposition is elaborated: what are the expected hardware and

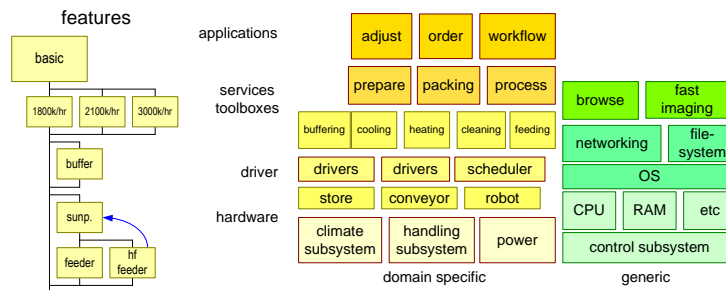


Figure 2.7: Identify Product Features and Technology Components

software components or building blocks, what are the dependencies between them? The main purpose of this step is to understand the potential commercial and technical modularity. From this modularity the synergy can emerge between products.

## 2.2.4 Make maps

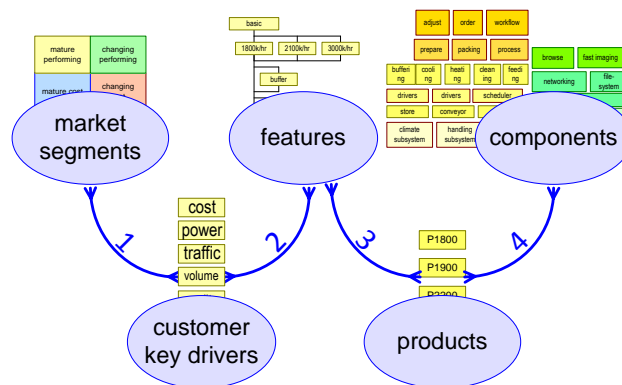


Figure 2.8: Mapping From Markets to Components

The first views have resulted in the identification of *market segments*, *customer key drivers*, *features*, *products*, and *components*. In this step the objective is to relate these views, e.g. *market segments* to *customer key drivers*,

*customer key drivers* to *features*, *features* to *products*, and *products* to *components*, see Figure 2.8. Each mapping can be many to many, for example different market segments can share the same key drivers, while every market segment has multiple key drivers.

<ul style="list-style-type: none"> <li>• Value for the customer</li> <li>• (dis)satisfaction level for the customer</li> <li>• Selling value (How much is the customer willing to pay?)</li> <li>• Level of differentiation w.r.t. the competition</li> <li>• Impact on the market share</li> <li>• Impact on the profit margin</li> </ul>
Use relative scale, e.g. 1..5 1=low value, 5 -high value
Ask several knowledgeable people to score
Discussion provides insight (don't fall in spreadsheet trap)

Figure 2.9: Example Criteria for Determining Value

## 2.2.5 Discuss value, synergy and (potential) conflicts

In general the wish list for features is longer than can be implemented in the first releases. We need more insight in the value of the different features to facilitate a selection process, as discussed in [25].

		— products →											
		P1800			P1900			P2200					
features ↓		satisfaction	customer	sales price	market share	satisfaction	customer	sales price	market share	satisfaction	customer	sales price	market share
	feeder	1	5	4		3	4	4		4	5	5	
	hf feeder												
	buffer	4	3	4		5	3	4		4	3	4	
	sunpower	2	2	1		2	2	1		2	2	4	

Figure 2.10: Determine Value of Features

Figure 2.10 shows the results of this selection process. Note that the discussion provides most of the value to the exploration team. The need to characterize and agree on the scoring forces the team to compare features and to articulate their value.

## 2.2.6 Create long term and short term plan

Practical constraints such as time and effort often determine our choices in synergy and the order in which we realize these choices. The exploration team has to translate their vision that has grown into a plan showing in what order it could be realized. Part of the plan will be short term: what do we do rather concrete in

the next few weeks or months? The long term plan visualizes the big picture of moving towards synergy: how do we envision that we will migrate to the synergistic situation? Note that again making the short term and long term plan serves the purpose to force the exploration team in this practical discussion.

## 2.3 Example of synergy

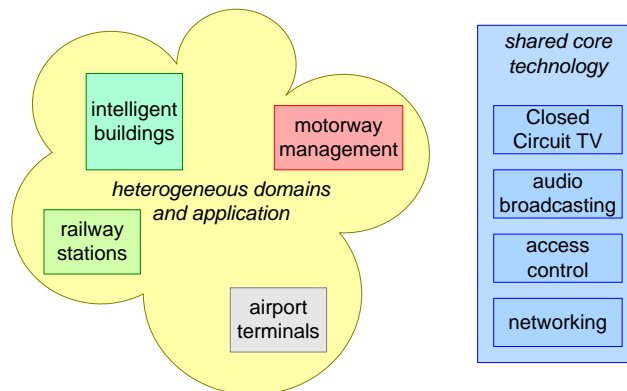


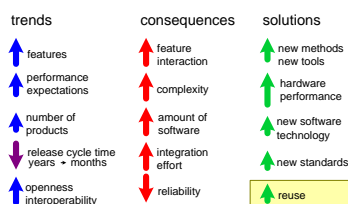
Figure 2.11: Example of synergy between heterogeneous markets

Figure 2.11 shows an example of a company serving 4 heterogeneous markets: intelligent buildings, motor way management systems, airport terminals, and train stations. This company performs projects in these 4 markets, providing Closed Circuit Televisions, access control, audio broadcasting, and the integration. The synergy is in the technical components, such as cameras and loudspeakers. The question was if there is also potential synergy in the integration, e.g. the networking, system control, and operator interfaces. For that purpose the key driver diagram in [24], was developed.



## Chapter 3

# Software Reuse; Caught between strategic importance and practical feasibility



### 3.1 Introduction

Many good reasons exist to deploy a reuse strategy for product creation, see figure 3.1. 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.

Reuse is deployed already in many product development centers. Brainstorming with architects involved in such developments about their experiences gives a very mixed picture, see figure 3.2 for the bad versus the good experiences.

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 3.3 shows a number of examples.

Reuse strategies can work successfully for a long time and then suddenly run into problems. Figure 3.4 shows the limitations of successful reuse strategies.

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

- + 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 3.1: Why reuse: many valid objectives

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: [19] [18] [20] [13] [22] [17]

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 3.2: Experiences with reuse, from counterproductive to effective

homogeneous domain	cath lab MRI television waferstepper
hardware dominated	car airplane shaver television
limited scope	audio codec compression library streaming library

Figure 3.3: Successful examples of reuse

struggle with integration/convergence with other domains	how to innovate?
TV: digital networks and media cath lab: US imaging, MRI	
poor/slow response on paradigm shifts	
TV: LCD screens cath lab: image based acquisition control	
software maintenance, configurations, integration, release	
MRI: integration and test wafersteppers: number of configurations	

Figure 3.4: Limits of successful reuse

## 3.2 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 3.5 and 3.6 about reuse.

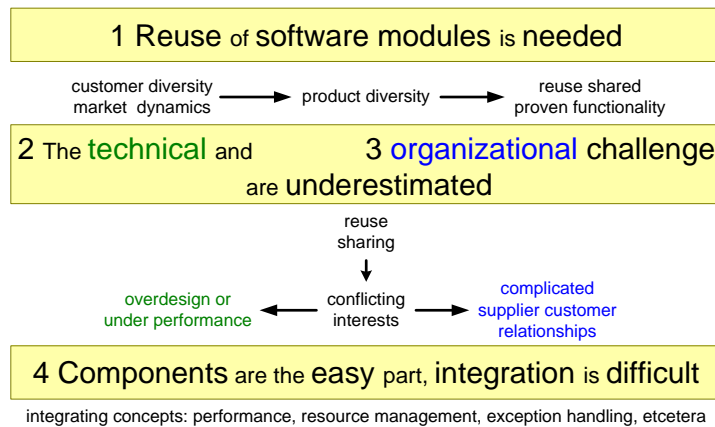


Figure 3.5: Reuse statements

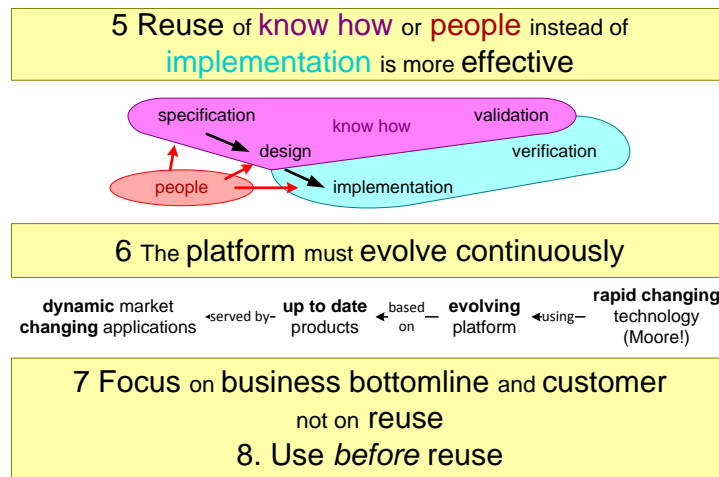


Figure 3.6: Reuse statements continued

### 3.3 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.

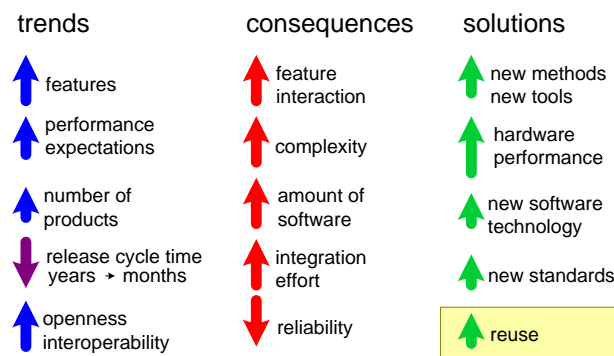


Figure 3.7: Reuse is needed ... as part of the solution

Figure 3.7 show these trends in the market in the left hand column, where the length of the arrow indicate the relative increase or decrease.

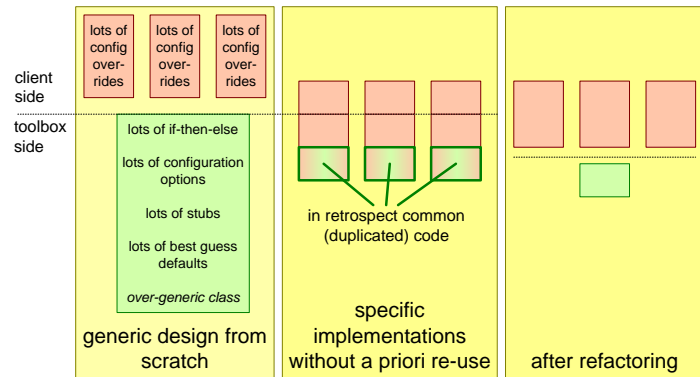
The consequence of the market trends for product creation are that more and more features start to interact and that the complexity increases. This is reflected in a string growth in the amount of software in products. The integration effort increases also. The combination of these factors threaten the 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.

### 3.4 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.



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

Figure 3.8: The danger of being generic: bloating

Figure 7.3 show an actual example of part of the Medical Imaging system [16], 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 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.

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 7.1 shows a number of causes for bloating.

One of the bloating problems is that bloating causes more bloating, as shown in figure 7.6. 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 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

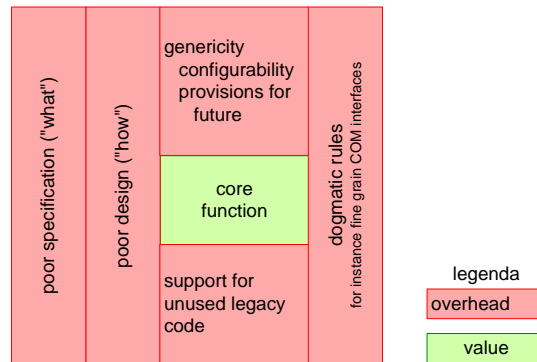


Figure 3.9: Exploring bloating

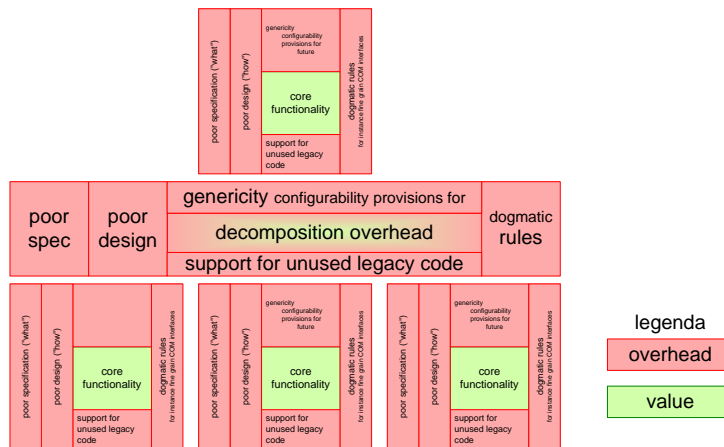


Figure 3.10: Bloating causes more bloating

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 hardware costs, loss of overview, et cetera.

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

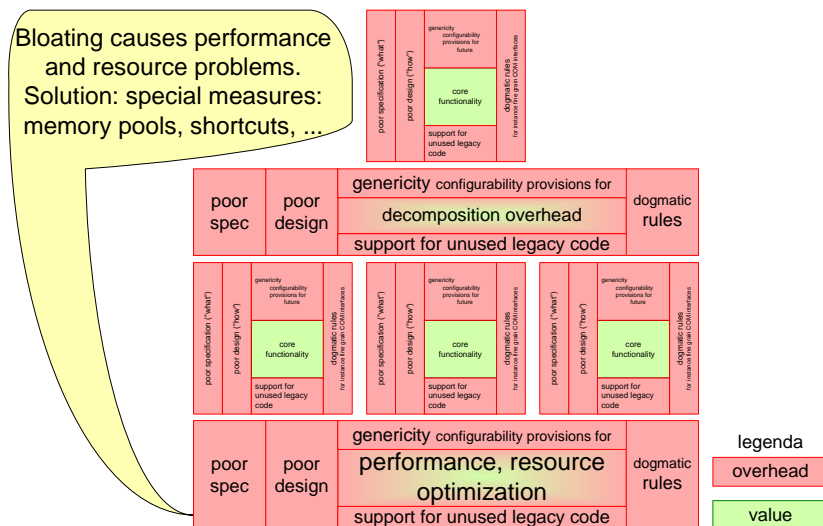


Figure 3.11: causes even more bloating...



### 3.5 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 3.12

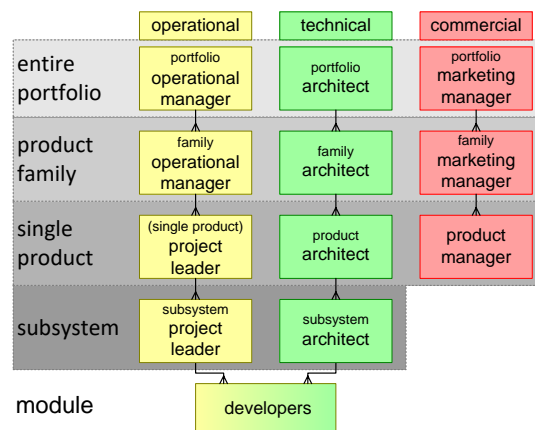


Figure 3.12: Conventional operational organization

The introduction of reuse has a big impact on this hierarchy in the operational organization of the PCP. Figure 3.13 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 3.14 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

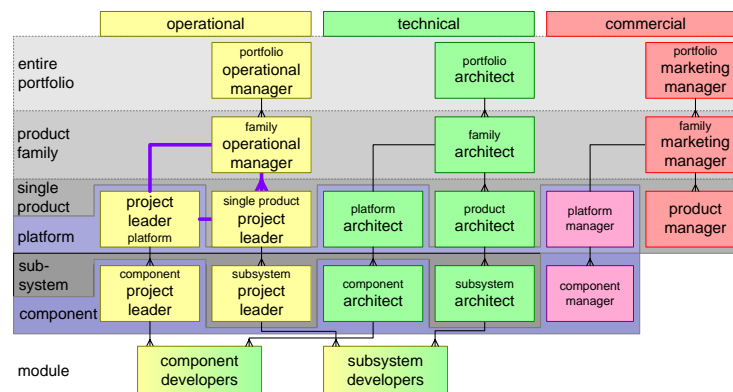


Figure 3.13: Modified operational organization

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 3.13 contains a few other peculiarities. 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 peculiarity 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 occurring 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.

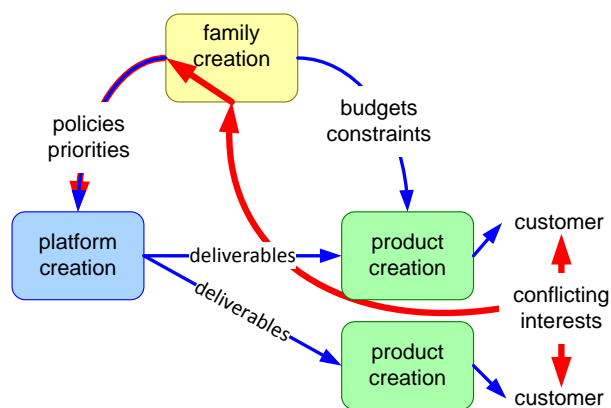


Figure 3.14: Conflicting interests of customers escalate to family level, have impact on platform, product creation teams benefit or suffer from the top down induced policy

## 3.6 Integration

Many people expect the architect to decompose, as mentioned in the explanation of "guiding how", while integration is severely underestimated, see figure 6.18. 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.

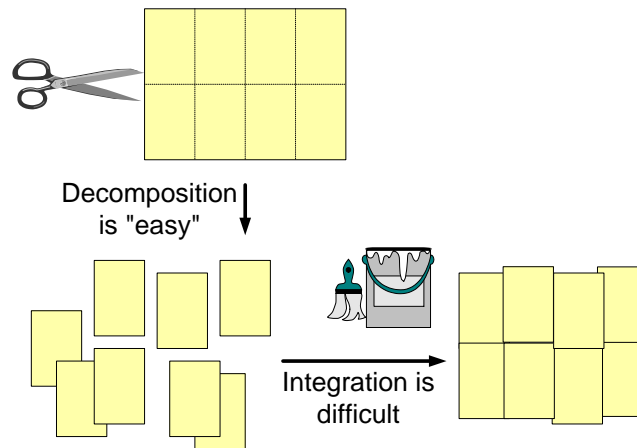


Figure 3.15: Decomposition is easy, integration is difficult

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 3.16 visualizes this process. The invisible problems cause a significant delay<sup>1</sup>.

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

<sup>1</sup>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.

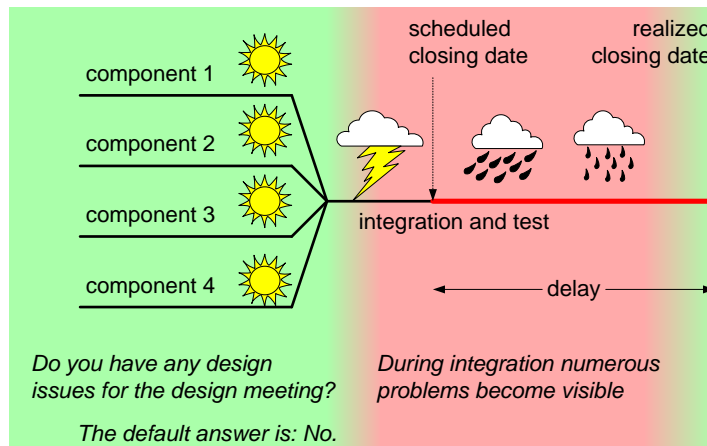


Figure 3.16: Integration problems show up late during the project, as a complete surprise

designers and engineers. Guiding the implementation is done by providing guidelines and high level designs for many different viewpoints. Figure 3.18 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 3.18 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 3.19 shows a blueprint of a full blown platform.

The blueprint shows a superset of what can be part of the platform. Figure 3.19 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 [13].

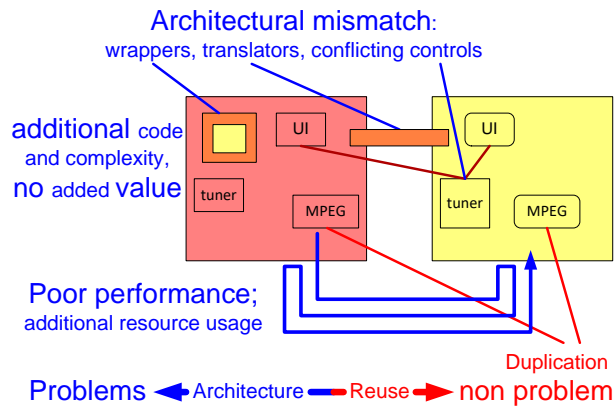


Figure 3.17: Integration of components from different sources is difficult due to the architectural mismatch

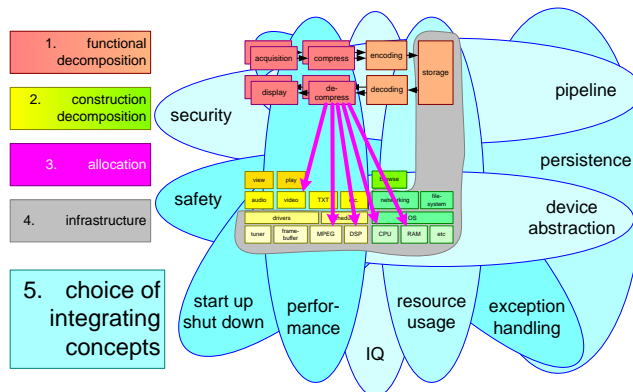


Figure 3.18: Integrating concepts

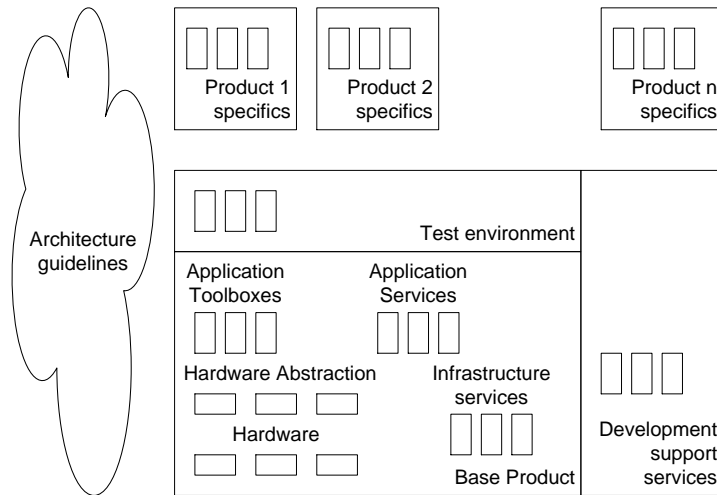


Figure 3.19: Platform block diagram

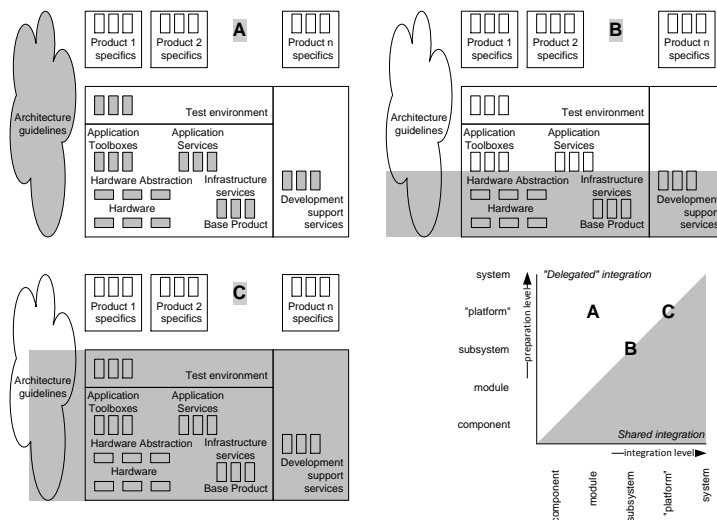


Figure 3.20: Platform types

### 3.7 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 6.19 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 [19]. In other words it is a miracle if a platform is stable, when both the supplying as well as the consuming side are not stable at all.

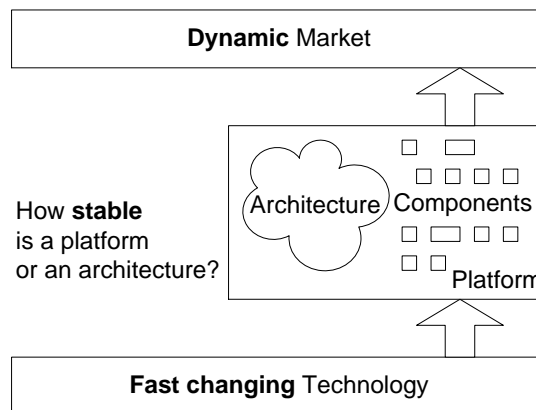


Figure 3.21: The outside world is dynamic

The evolution of a platform is illustrated in figure 6.22 by showing the change in the Easyvision [16] 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.

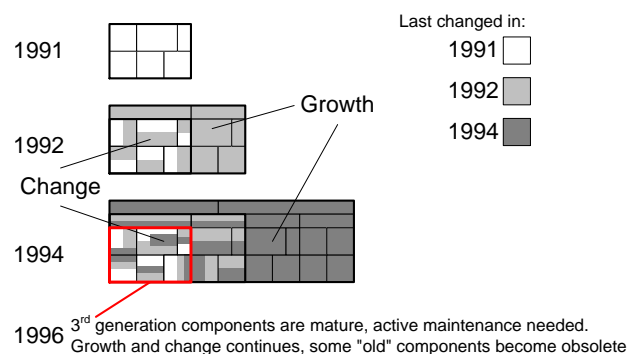


Figure 3.22: Platform evolution (Easyvision 1991-1996)



### 3.8 Reuse of know how

The CAFCR model [21] 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.

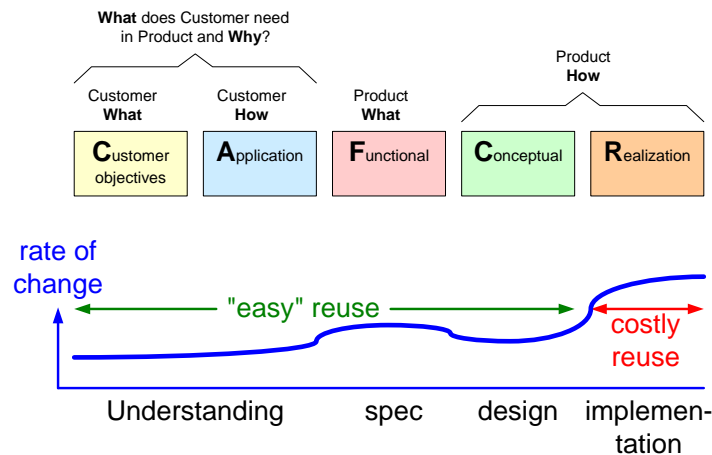


Figure 3.23: Reuse in CAFCR perspective

Figure 3.23 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.

### 3.9 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 3.24.

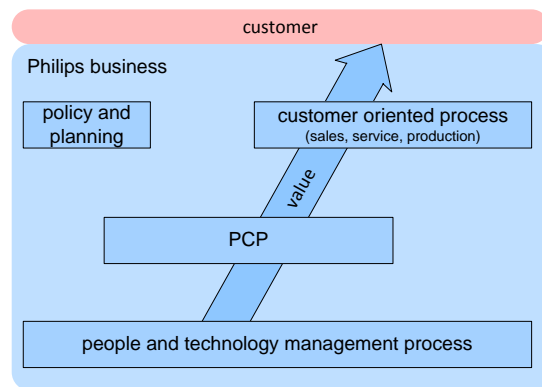


Figure 3.24: Simplified decomposition of the business in 4 main processes

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.

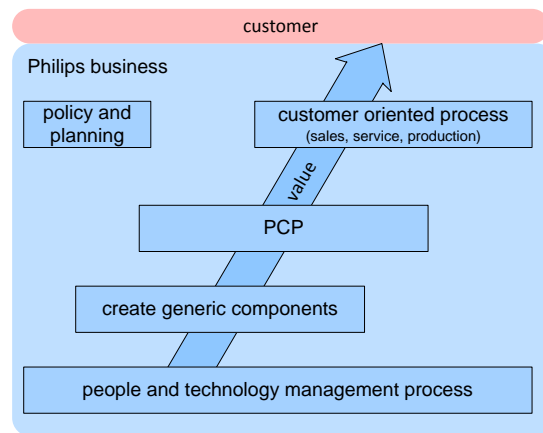


Figure 3.25: Modified Process Decomposition

The simplified process description given in figure 3.24 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 3.25 shows the modified process decomposition (still simplified of course) including this additional process "Generic Something Creation Process".

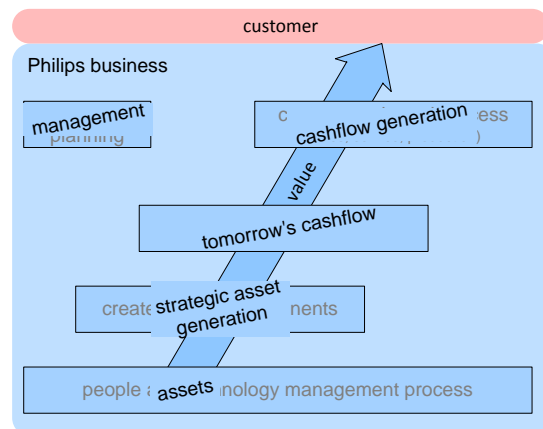


Figure 3.26: Financial Viewpoint on Process Decomposition

Figure 3.26 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

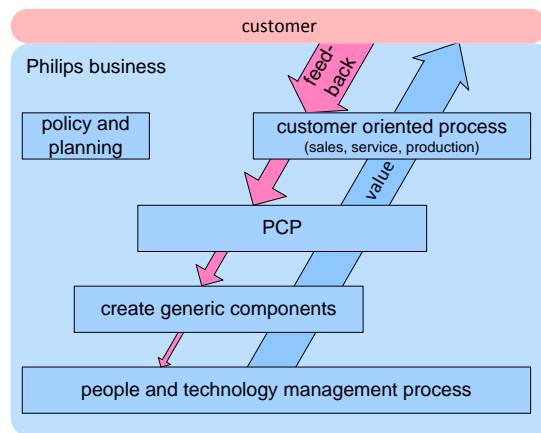


Figure 3.27: Feedback flow: loss of customer understanding!

and consequently a longer feedback chain as well. This is shown in figure 3.27. 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 3.28.

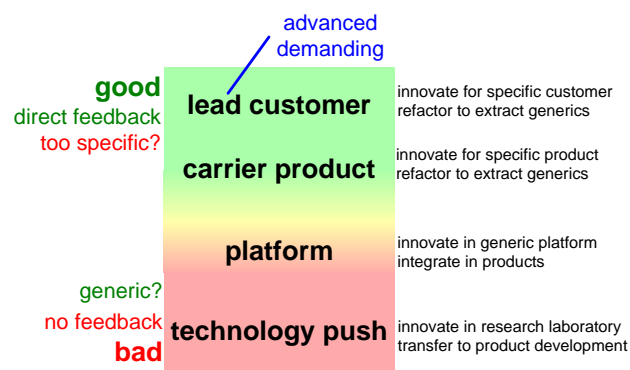


Figure 3.28: Models for SW reuse

### 3.9.1 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.

### **3.9.2 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.

### **3.9.3 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 3.29.

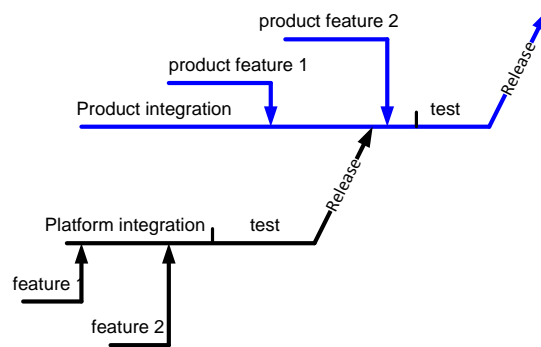


Figure 3.29: The introduction of a new feature as part of a platform causes an additional latency in the introduction to the market.

### 3.10 Use before reuse

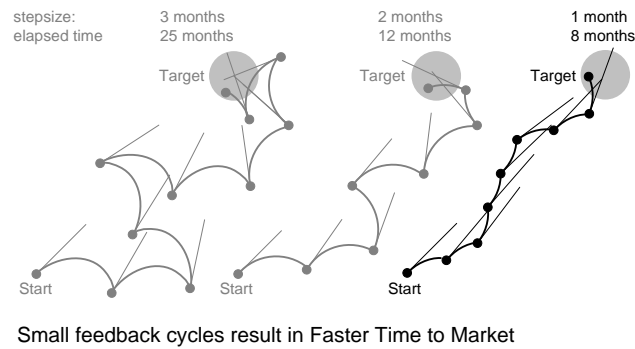


Figure 3.30: Feedback (3)

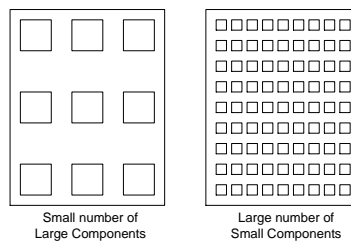
Understanding of the problem as well as the solution is key to being effective. Learning via feedback is a quick way of building up this understanding. Waterfall methods all suffer from late feedback, see figure 7.14 for a visualization of the influence of feedback frequency on project elapsed time.

- Does it satisfy the needs?
  - performance
  - functionality
  - user interface
- Does it fit in the constraints?
  - cost price
  - effort
- Does it fit in the design?
  - architectural match
  - no bloating
- Is the quality sufficient?
  - multiplication of problems
  - or multiplication of benefits

Figure 3.31: Use of software modules enables validation before Reuse

## Chapter 4

# Aggregation Levels in Composable Architectures



### 4.1 Problem description

This article is focusing on "composable architectures". Composable architectures are designed for a single application domain, enabling the composition of products of which the definition is still evolving or hidden in the future.

A crucial design question is: *What is the desired granularity of the design, what are useful abstractions?* The granularity of the design is directly related to the question: *What are the appropriate aggregation levels for composition and integration?*

Most Product Creation Processes are based on a single dominating decomposition and integration model. This oversimplification causes many problems for development.

This article describes an approach based on multiple viewpoints, matching the wide variety of concerns involved. Per viewpoint heuristics are given.

Application of a multiview approach requires customization of viewpoints and concerns. In general this means identification of the most relevant, important of critical issues, which are used to select a small manageable amount of viewpoints as main focus.



## 4.2 Views on Aggregation

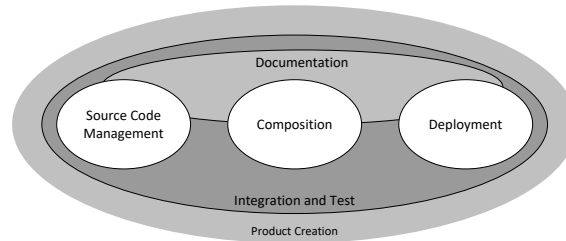


Figure 4.1: Venn diagram showing the overlap between Viewpoints on Aggregation Levels

Figure 4.1 shows a Venn diagram with 5 viewpoints with respect to aggregation levels, in the overall context of Product Creation. For every viewpoint the dominating concerns are mentioned in table 4.1 and the related aggregation levels or entities in table 4.2.

Viewpoint	Concerns
Documentation	Requirements, Specification, Design, Transfer, Test, Support
Source Code Management	Storage, Management, Generation
Composition	System, Subsystem, Function, Application
Deployment	Releasing, Distribution, Protection, Update, Installation, Configuration
Integration and Test	Confidence, Problem Tracking

Table 4.1: *Concerns per viewpoint*

All entities in Documentation, Repository, Composition and Deployment are relevant for the Integration and Test viewpoint.

## 4.3 Documentation

Many types of documentation are required when building Product Families by means of Composable Architectures. The granularity issues with respect to documentation are described in [12].

The aggregation levels for documentation are shown in table 4.2. Figure 4.2 visualizes the documentation concerns. For every level relevant documents should be produced, with respect to the *what* (requirements, specifications), *how* (design), *transfer* (to Customer Oriented Process), verification (test) and *how-to* (support to

Viewpoint	Entities
Documentation	Product Family, Product/System, Function/Feature, Subsystem, Component, Building Block, Module
Source Code Management	Package, File
Composition	Product, Executable, Dynamic Library, Component
Deployment	Distribution Medium ("CD"), Unit of Licensing ("SW key"), Package, Patch, Configuration data
Integration and Test	Test Configurations, Intermediate Integration results

Table 4.2: Aggregation Levels or Entities per viewpoint

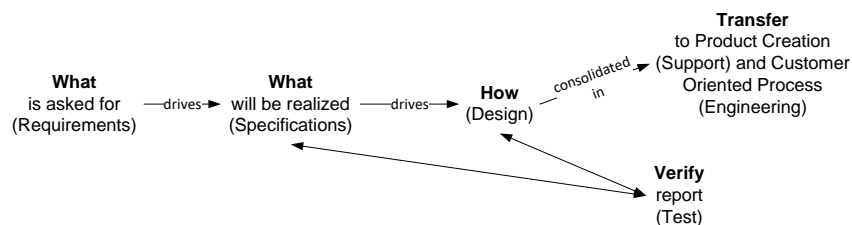


Figure 4.2: Visualization of documentation concerns

use reusable assets in creation of products). In *what* and *how* documents a selected amount of *why* need to be present.

The documentation structure will evolve in time. This evolution requires explicit refactoring steps in the product family lifecycle. The *why* and to a lesser extent the *what* will be factored out, because this information is more stable and therefore re-useable than the *how*. Part of the information will move "upward" in the aggregation level stack: generic patterns become clear, which are consolidated as abstractions on an higher aggregation level.

## 4.4 Source Code Management viewpoint

The elementary description of the system is in the source code. This source code is stored in a structured way in a repository, see figure 4.3. There is no hard requirement that the source code structure maps one-to-one on semantic entities in the composition world. However a one-to-one mapping helps in maintaining overview and understanding.

The main concerns in this view have to do with source code management:

- storage and accessibility of all source code

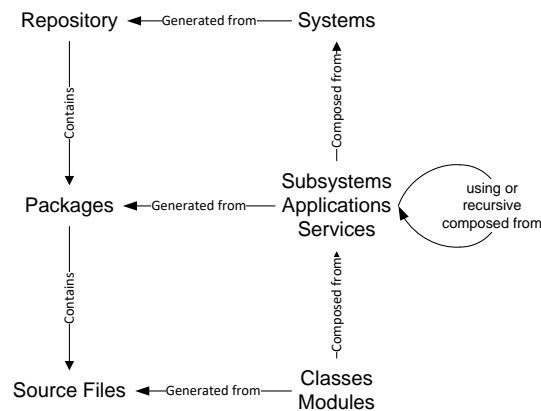


Figure 4.3: The source code is stored in files in a repository. The unit of structuring is called a package. These source code aggregation levels get a more semantic meaning when being used.

- version management; complete traceability of all versions and changes
- ownership for performance, quality and maintenance

The most widely used unit for management and storage of source data is **file**.

Source code in this context means all *original* formal descriptions, such as C, C++, include, text, data, make et cetera files. *Original* means that generated C-code does not belong to the source code, the data used for generating this code does belong to the source code.

The provide and require interface descriptions belong to the source code according to this definition, as do IDL interface definitions. For example see the KOALA component model as described in [26]. Generic subsystem configuration data defining the composition also belong to the source code.

Most source code need to be transformed in computer oriented intermediate formats before it can be used run time. The build step (compilation, building et cetera) required for this transformation may influence the repository structure. A well defined compile time dependency structure is desirable to enable a predictable composition step.

Table 4.3 shows the typical sizes, anno 2000, of source code repositories. The size is expressed in *lines of code* (loc). Historical data, see cost models in [3] and [1] shows a remarkable constant relationship between lines of code and the required manpower to create and maintain the software. The observed productivity in the Medical Imaging case study was ca. 10 kloc per manyear. Taking this number for a zero-order approximation the size of entities can be transformed in effort.

Entity	Typical size loc	packages
repository	1M-10M	10-100
package	10k–100k	
file	100-1k	

Table 4.3: *Typical Sizes of SW for Aggregation Levels*

This simple table illustrates a number of very essential design criteria, in relation to granularity of management.

Rules of thumb for typical file sizes are:

- Files should be larger than 100 loc;  
The overhead per file and the "value" per file must be balanced.
- Files should be less than 1000 loc;  
Large files reduce the overview within the module. Larger files are an indication for a lack of modularity.

The number of packages in the repository is mostly restricted by usage and testing configuration management concerns. A fine granularity with respect to packages (subsystems, applications or services in the composition view) enables a fine grained and powerful composition. Coarse granularity of packages means that more code is a priori bundled, constraining the freedom of the composer. The downside of fine granularity is a combinatorial explosion of the amount of configurations.

From more pure source code point of view the considerations for package size are:

- at least 10 files per package;  
Packaging files or modules generates some overhead in usage and management. The value of this packaging must be substantial to offset this additional overhead.
- at most 100 kloc per package to maintain overview;  
For unambiguous package-ownership and sufficient overview.

## 4.5 Composition viewpoint

Composition involves glueing together and configuring available components. The result of the composition process are "executable" entities such as components and plug-ins and more conventional executables and dynamic link libraries.

The granularity of these entities determines at the one hand the deployment flexibility at the other hand it determines the amount of testing and configuration management work.

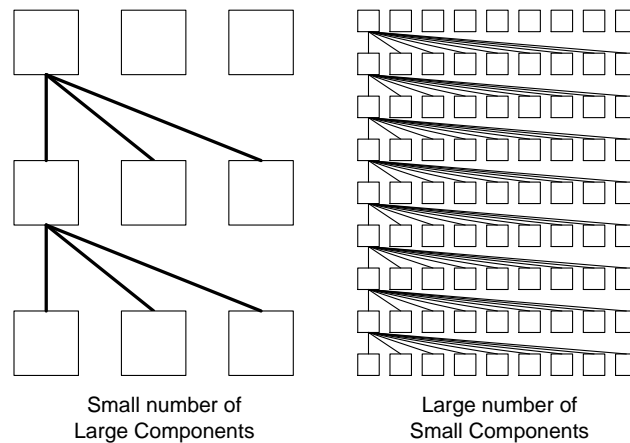


Figure 4.4: Coarse versus Fine grained with respect to the number of connections and relations; 9 large Components with 18 Connections, 81 small Components with 648 Connections

The number of relations between components is roughly in the order of  $n^{1.5}$ . Table 4.4 shows the number of components and the number of connections between them. The number of desired architects is derived from the number of connections by means of zero<sup>th</sup> order model. The "capacity" of an architect, the number of relations kept consistent and balanced by one architect, is used to determine the required number of architects:

$$NumberOfArchitects = NumberOfConnections / Capacity$$

A somewhat more realistic model takes into account that large components will have more complex connections with other components than small components. Table 4.5 shows the same model with an additional *weight* factor to model the complexity of the connection. The weight curve applied is rather arbitrary, it reflects the experience of the author.

#### 4.5.1 Optimal granularity for composition

The simple models in tables 4.5 and 4.4 make it immediately clear that a large quantity of components is undesirable. Assuming a total crews of circa 100 developers (which corresponds with today's multi-million lines of code repositories) it is reasonable to have 10 architects. The optimal number of components is than in the order 20 to 40.

Capacity of architects $c$		10	20	40
Number of components $n$	Number of relations $r = n\sqrt{n}$	Number of Architects $a = r/c$		
2	3	0	0	0
4	8	1	0	0
10	32	3	2	1
20	89	9	4	2
40	253	25	13	6
100	1000	100	50	25
300	5196	520	260	130
1000	31623	3162	1581	791

Table 4.4: *The relation between the number of components and the required number of architects, zero order model*

The above reasoning is entirely macroscopic, calibrated with some typical Philips products. In specific cases plenty of reasons can exist which enable a higher number of components. For instance:

- presence of a stable reference model
- variation of components hidden behind an effective abstraction
- tangible and therefore understandable, predictable domain

## 4.6 Field Deployment viewpoint

The granularity in the field deployment is determined by pragmatics of the Customer Oriented Process [17]. These pragmatics can be further decomposed, see table 4.6.

Conventional embedded products do not have any field deployment activity, these systems run out of the box. The increasing availability of network connectivity enables field updates, with all related configuration management consequences.

At this moment no heuristics are available for the granularity with respect to the drivers in table 4.6.

## 4.7 Integration and Test viewpoint

The real challenge in composable architectures is the integration and testing. Building small building blocks is the easy part, getting them to work correctly together with many other building blocks is more difficult.

Capacity of architects $c$			10	20	40
Number of components $n$	Number of relations $r = n\sqrt{n}$	weight $w$	Number of Architects $a = (r * w)/c$		
2	3	12	3	2	1
4	8	9	7	4	2
10	32	4	14	7	3
20	89	2	22	11	5
40	253	2	39	19	10
100	1000	1	114	57	28
300	5196	1	534	267	133
1000	31623	1	3176	1588	794

Table 4.5: *The relation between the number of components and the required number of architects, first order model*

- granularity of sellable features and services
- lifecycle support
- internal logistics and production process

Table 4.6: *Decomposition of Field Deployment granularity drivers*

A bottom up test philosophy, where every building block is verified in isolation helps, because it reduces the number of difficult to trace errors during integration. Bottom up testing needs to be complemented by an integration philosophy.

The time needed for verification of a building block depends exponentially or worse on its size. The combinatorial explosion of possible (and useful) states limits the optimal size of elementary building blocks. The typical size for verifiable modules is between 100 loc and 10 kloc. In section 4.5 the optimal number of components is derived to be between 20 and 40. For multi-million loc products a typical component will exceed the size of being bottom up verifiable.

Figure 4.5 shows the cost of bottom up testing as function of the size and the duration of the complementary integration also as function of the size. Note that the integration duration more or less increases linear, while the size increases exponential. The simple explication for this is that every integration step halves the number of modules to be integrated, the schedule looks like an horizontal binary tree. In other words the duration is logarithmic with the total size.

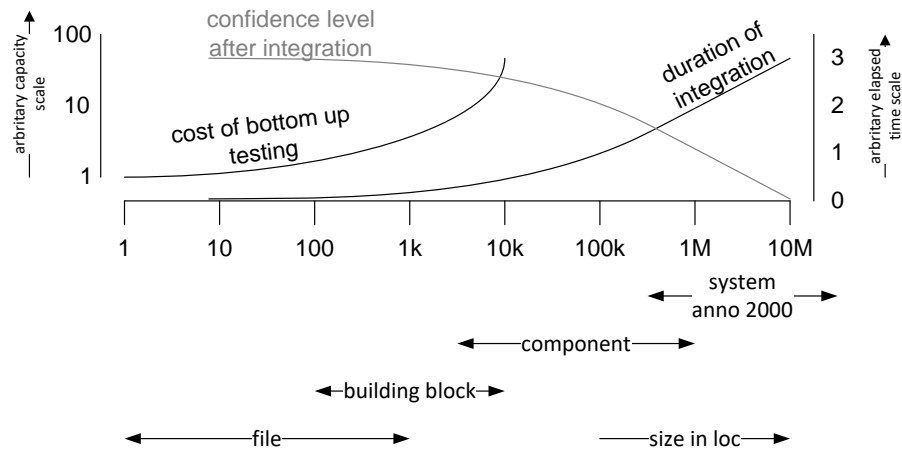


Figure 4.5: Integration and testing as function of size

Integration is a non-exhaustive activity. Best case the most relevant (from usage and test risks perspective) areas are touched. This means that the level of confidence obtained by integration decreases with increasing size.

An acceptable level of confidence is only reached by a combination of bottom up testing, integration testing and intermediate common sense verification steps in between.

## 4.8 Acknowledgements

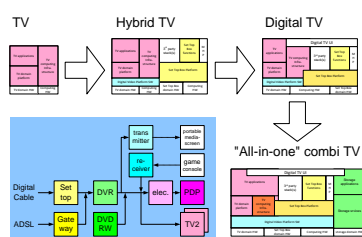
This paper has been written as part of the "composable project". The project members are: Pierre America, Hans Jonkers, Jürgen Müller, Henk Obbink, Rob van Ommering, William van der Sterren, Jan Gerben Wijnstra and Gerrit Muller. It has been discussed within the team, and the team contributed significantly to the contents.

Jürgen Müller suggested several improvements with respect to flow, consistency and balance. Wim Vree indicated multiple improvements, amongst others "local terminology and acronyms", which have either to be avoided or explained.



## Chapter 5

# From Legacy to State-of-the-art; Architectural Refactoring



### 5.1 The problem

#### 5.1.1 Market trends

Consumer Electronics Products are a large variety of products, which have evolved from straightforward electronic devices, such as radios, into complex software intensive systems. Figure 5.1 shows a typical set of today's audio and video products.

Technological advances and business opportunities result in a convergence of separate worlds. The worlds of *telecommunications*, *computers* and *consumer electronics* are converging, see figure 6.4.

This convergence means that functions from the different domains are integrated in new types of appliances. These appliances are optimized towards the intended use. User, form factor, function and environment all together determine what an optimal appliance looks like. The wide variety in users, form factors, functions and environments requires a very rich variety of appliances.

Figure 6.5 shows at the left hand side a small subset of existing devices belonging in one of the three domains. More to the right some of the form factors are shown,



Figure 5.1: Today's Audio Video Consumer Products

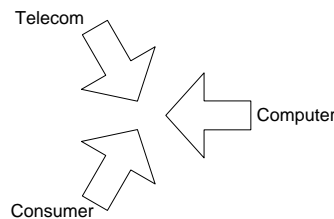


Figure 5.2: Trend: Convergence of separate worlds

while the right hand side shows some of the environments. The number of useful combinations of functions, form factors and environments is nearly infinite!

In this presentation video entertainment will be used as the application area. Figure 5.4 shows a typical diagram of the set up of video products in our homes. We see products to connect with the outside world (set top box), storage products (Video Cassette Recorder abbreviated as VCR), and conventional TV's and remote controls, which are the de facto user interface.

This chain of video products is slowly evolving, as depicted in figure 5.5. In the past a straightforward analog chain was used. The elements in this chain are stepwise changed into digital elements. The introduction of the Large Flat TV's breaks open the old paradigm of an integrated TV, which integrates tuner and related electronics with the monitor function.

In the near future many more changes can be expected, such as the introduction of the Digital Video Recorder (DVR), a gateway to alternative broadband solutions, wireless inputs and outputs, home networks enabling multiple TV's.

The function allocation for this last stage of figure 5.5 and the network topology can be solved in several ways. Figure 5.6 shows four alternatives, based on a client-server idiom.



Figure 5.3: Integration and Diversity

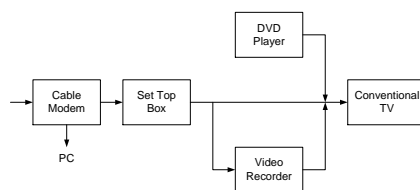


Figure 5.4: Today's Video Products

The expectation is that all alternatives will materialize, where the consumer chooses a solution which fits his needs and environment.

These alternatives require different packaging of functions into products, as shown in figure 5.7.

### 5.1.2 Technology trends

The major trend for electronic devices is Moore's law, roughly stating that the available amount of transistors in an integrated circuit doubles every 18 months. Devices based on IC's follow this trend. Figure 6.7 shows the growth of the amount of memory in TV's.

The amount of software in products, measured as lines of code, more or less follows Moore's law. Unfortunately the software engineering discipline did not proceed at the same rate, which is reflected by a fault metric expressed as fault per thousand lines of code, varying between 1 for very rigid organized producers, to

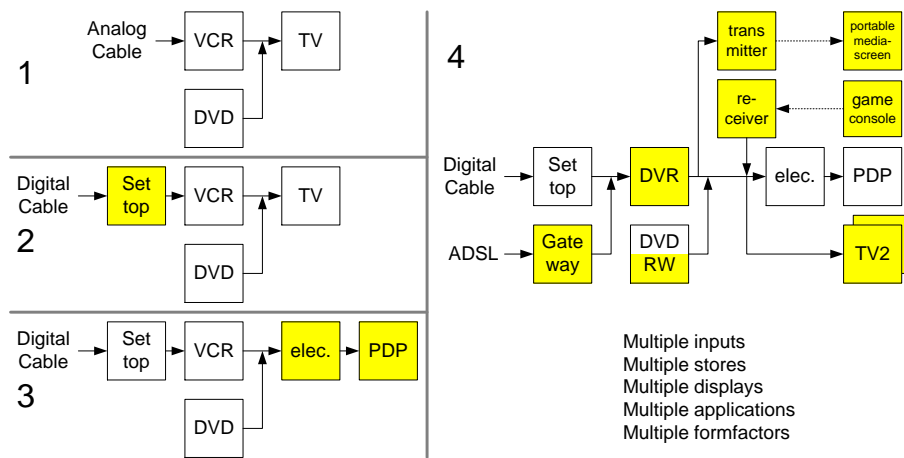


Figure 5.5: Evolution of Video Products

10 or more for ad hoc products. Typical values for CE type products is 3 faults per 1000 lines of code. See figure 5.9 for typical values as function of the year.

The increase of the amount of software causes many problems:

- Increase of development cost
- (Non) availability of skilled engineers
- Increase of development time, and hence time to market
- Decrease of product reliability

Reuse is often presented as **the solution** for all problems mentioned above. Experience learns that quite the opposite happens in many cases, see [13], the challenge of executing a successful reuse program is often severely underestimated.

The most common root-cause of reuse failure is the mistake to see reuse as a goal rather than a means.

### 5.1.3 Example Digital Television

An example of a new product is a digital television, which is the merger of a set top box and a television. This television can directly connect to a digital cable infrastructure and offer services provided by cable or content providers.

One way of realizing such a system is to declare the reuse of existing software, integrate all this software on a single hardware platform and to support this hardware platform factor out the “lower” software layers. Figure 5.11 shows the simplistic architecting to achieve this merger.

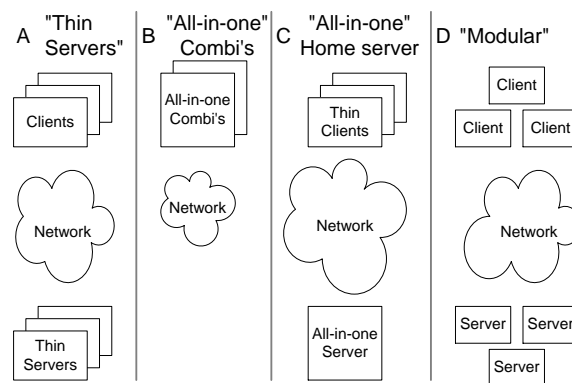


Figure 5.6: Distribution Scenario's

Figure 5.12 shows the rationale behind the reuse of existing software packages. The cumulative effort of the software involved exceeds 500 manyears.

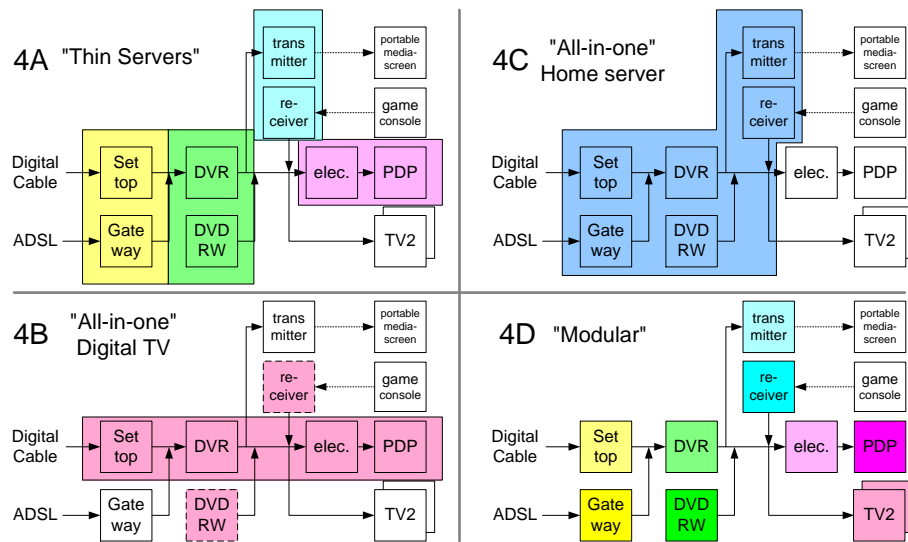


Figure 5.7: Product Packaging Options

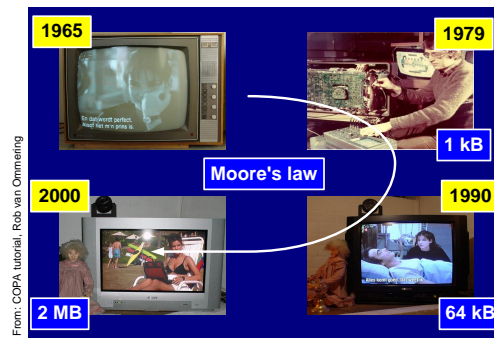


Figure 5.8: Moore's law

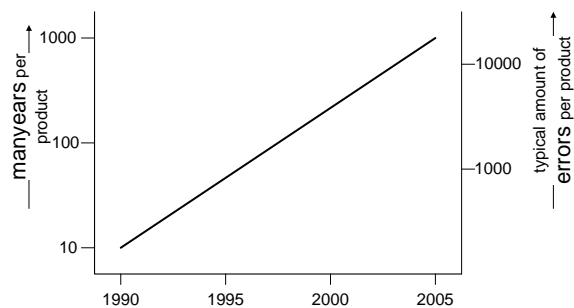


Figure 5.9: Problem: increasing SW size, decreasing reliability?

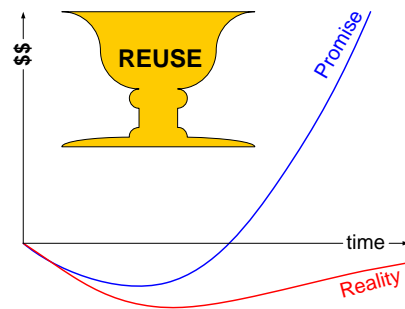


Figure 5.10: The Holy Grail: Reuse

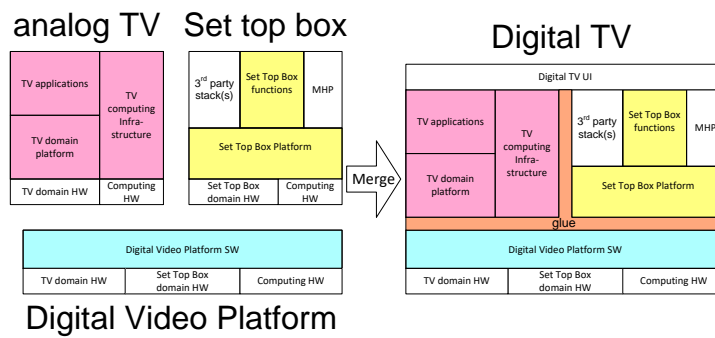


Figure 5.11: Simplistic Architecting: Digital TV

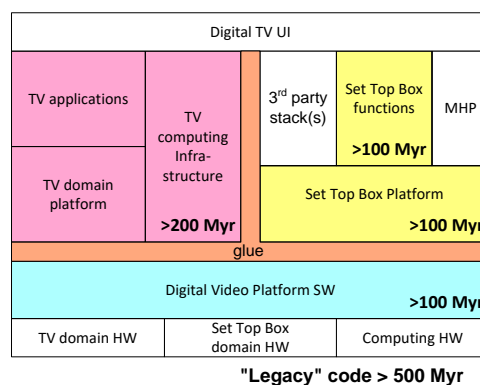


Figure 5.12: Available Code Assets

## 5.2 Architectural Refactoring

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.

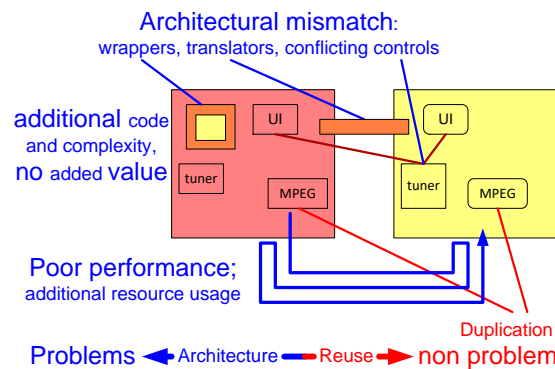


Figure 5.13: Merge problems

The proposed solution to this set of problems is **architectural refactoring**. Architectural refactoring is an incremental approach, putting a lot of emphasis on feedback. Two major criteria to get feedback on are:

- How well does the current architecture support today’s product needs?
- How well will the architecture evolve to follow the market dynamics?

In every increment to the market both concerns should be addresses, which translates in clear business goals (product, functions, value proposition) **and** clear refactoring goals fitting in a limited investment. The refactoring goals should be based on a longer term architecture vision, see 5.14.

Examples of Refactoring goals can be seen in figure 5.15. These refactoring goals should be sufficiently “SMART” to be used as feedback criterium.

*Note: many refactoring projects spend lots of effort, while critical review afterwards does not show any improvement. Often loss of goal or focus is the basis for*



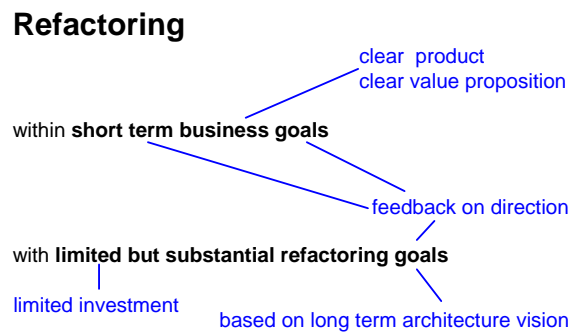


Figure 5.14: Solution: Architectural Refactoring

*such a disaster.*

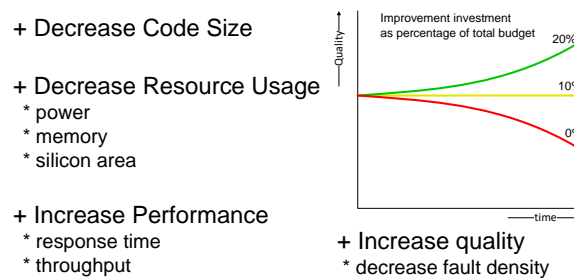


Figure 5.15: Example of Refactoring Goals

Architectural refactoring looks at all architectural aspects, from functions and structure to selection of mechanisms and technologies. Code refactoring, well known from extreme programming [2], plays a role at a much more microscopic level. See figure 5.16 which shows both ways of refactoring side by side. Some code refactoring requires an update of the architecture. At the other hand architectural changes quite often have a significant software impact.

## 5.2.1 Prerequisites for effective architectural refactoring

### Frequent feedback

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

### Awareness of dynamics

The world is highly dynamic, the markets and applications change rapidly, while the famous law of Moore shows the incredible speed of technological devel-

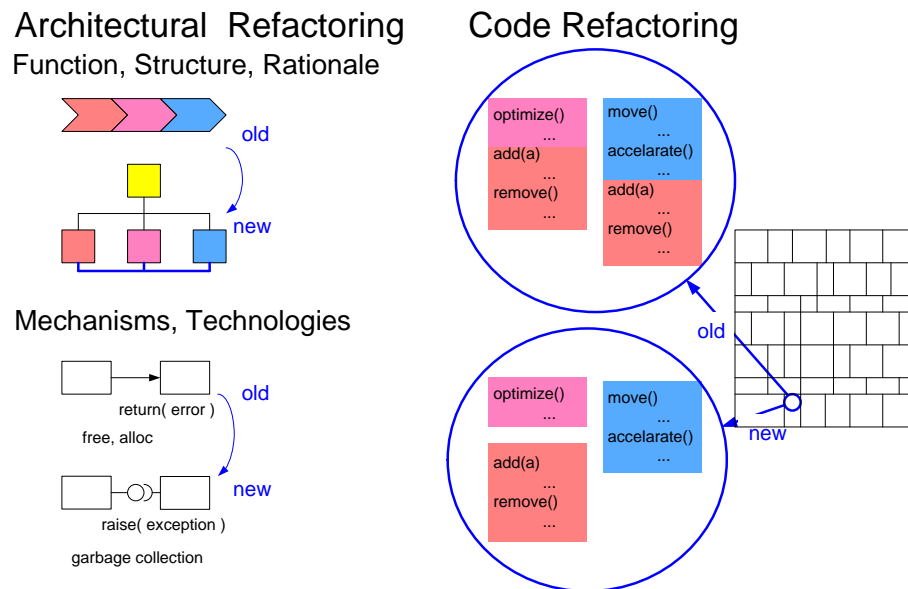


Figure 5.16: Architectural and Code refactoring

opments. Unfortunately most people believe in stability and are biased towards stabilizing architectures. Architectures and their implementations are sandwiched between the fast moving market at one side and technology improvements at the other side. Since both sides change quite rapidly, the architecture and its implementation will have to change in response, see figure 6.19.

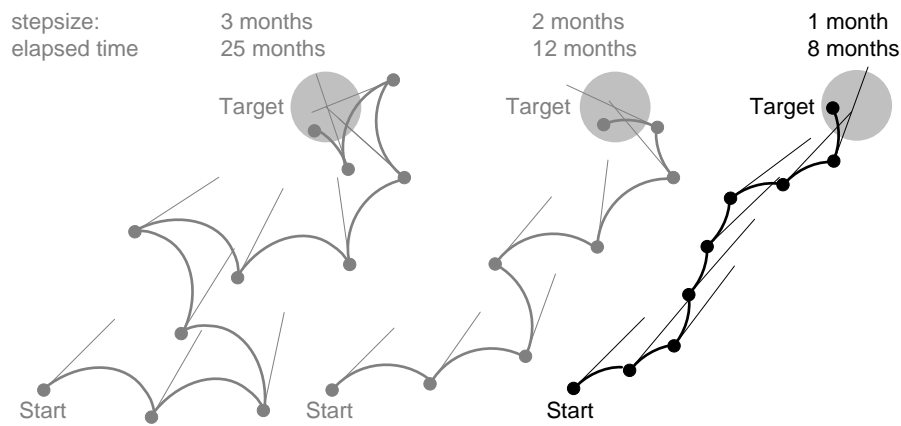
The evolution of a platform is illustrated in figure 6.22 by showing the change in the Easyvision [16] platform in the period 1991-1996. It is clearly visible that every generation doubles the amount of code, while at the same time half of the existing code base is touched by changes.

### Long Term Vision

In order to set refactoring goals it is useful to have a long term vision on the architecture. Such a long term vision may be quite ambitious. The ambition of the vision will be balanced by the pragmatics of short term business goals and limited investments in improvement.

Figure 5.20 shows an example of a long term vision, where a framework is foreseen, which decouples 6 design and implementation concerns:

- applications
- services
- personalization



Small feedback cycles result in Faster Time to Market

Figure 5.17: Frequent feedback results in faster results and a shorter path to the result

- configuration
- computing infrastructure
- domain infrastructure

The actual implementation will not have such a level of decoupling for a long time, the penalty in effort, resource usage and many other aspects will be prohibitive for a long time. Nevertheless the decoupling will become crucial if the variety of products is really very large and dynamic.

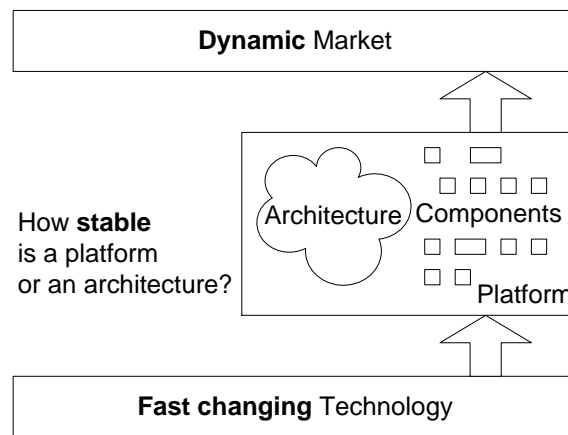


Figure 5.18: Myth: Platforms are Stable

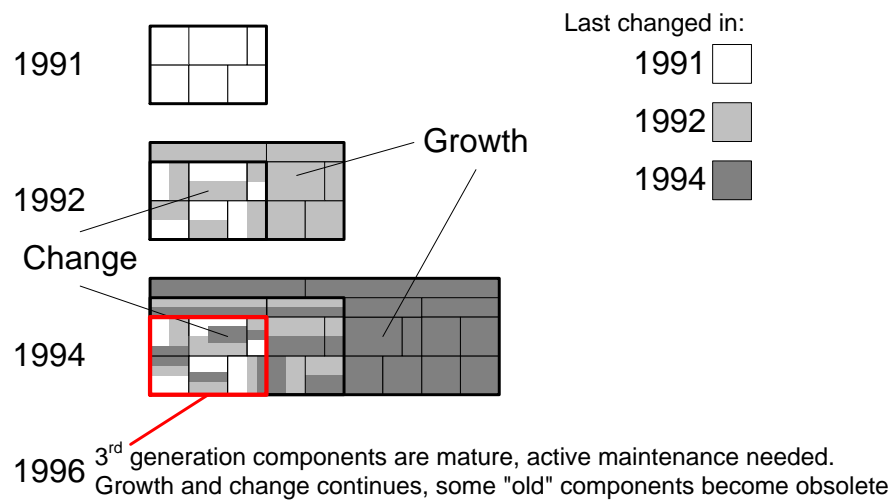


Figure 5.19: Platform Evolution (Easyvision 1991-1996)

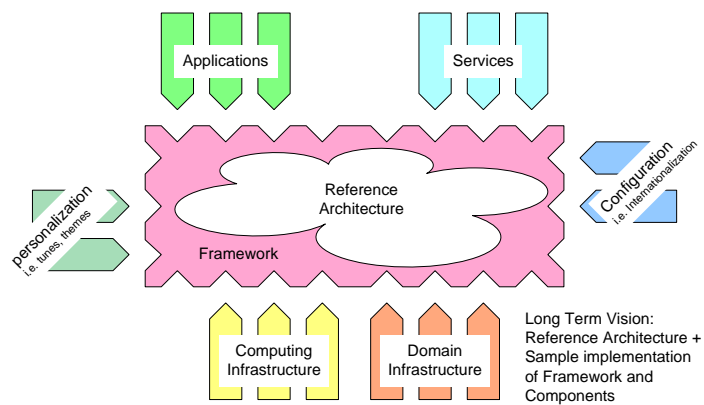


Figure 5.20: Example Long Term Vision

## 5.3 Conclusion

Figure 5.21 shows how **not** to work towards the future:

- Don't merge blindly
- Don't a priori declare SW to be reusable

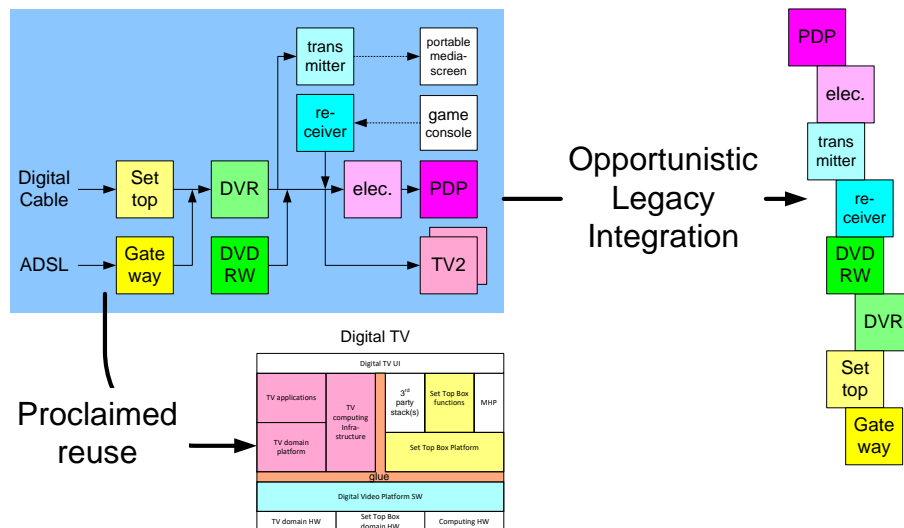


Figure 5.21: Don't do

While figure 5.22 illustrates architectural refactoring, applied on the example of a digital television. The steps taken here are:

*From TV to Hybrid TV.* The conventional TV is refactored to use a more modern HW platform, while the lower layer is factored out. The set top box is physically integrated in the television, while at software level both applications are pragmatically interfaced.

*From Hybrid TV to Digital TV.* More hardware is shared between the TV part and the set top box part of the system, with as refactoring goals: reduction of resource usage and enabling a more harmonized user interface. The set top box platform is redesigned to make this possible.

*From Digital TV to "All-in-one" TV.* The TV computing infrastructure is simplified (reduce lines of count), while the next "legacy" application is merged in: storage.

## 5.4 Acknowledgements

Lex Heerink patiently listened to the presentation and provided valuable feedback.

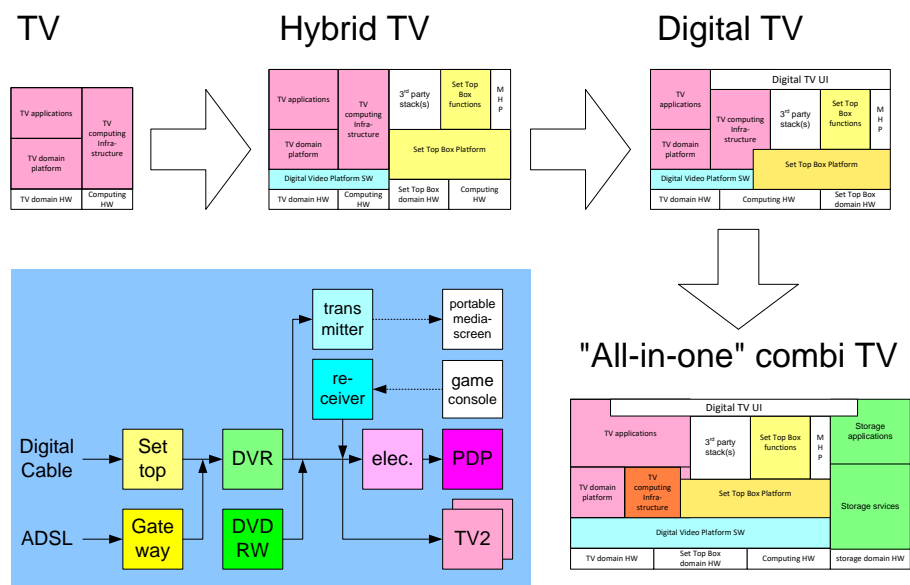
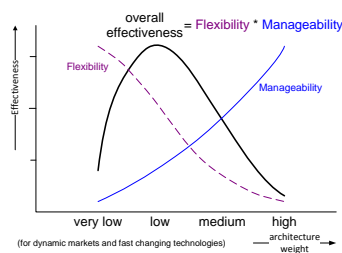


Figure 5.22: Conclusion: Refactoring the Architecture is a must

## Chapter 6

# Light Weight Architecture: the way of the future?



### 6.1 Introduction

Architecture is the combination of the know how of the solution (technology) with understanding of the problem (customer/application). The architect must play an independent role in considering all stakeholders interests and searching for an effective solution. The fundamental architecting activities are depicted in figure 6.1.

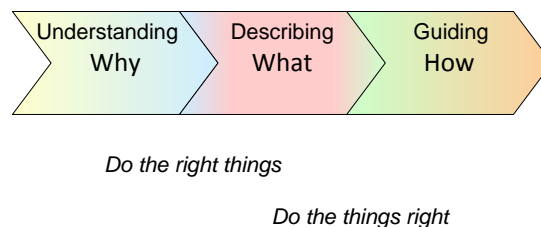


Figure 6.1: What is Architecture?



*Do the right things* is addressed in section 6.2. *Do the things right* is addressed in section 6.3. The weight of an architecture is discussed in section 6.4. This structure of the presentation is visualized in figure 6.2.

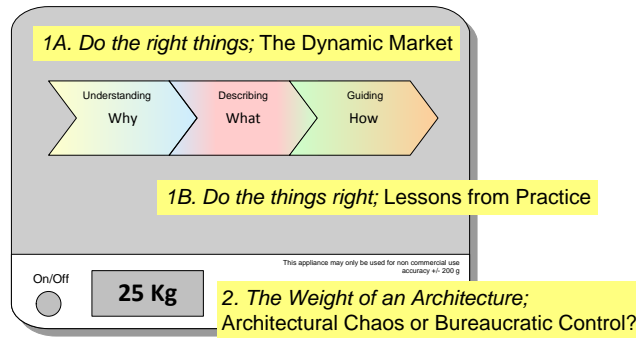


Figure 6.2: Table of Contents

## 6.2 Do the right things; The Dynamic Market

Philips Semiconductors (PS) plays a part in a longer value chain as depicted in figure 6.3. Typical the components of PS, such as single chip TV's, are used by system integrators, which build CE appliances, such as televisions. These appliances can be distributed via retail channels or via service providers to end consumers.

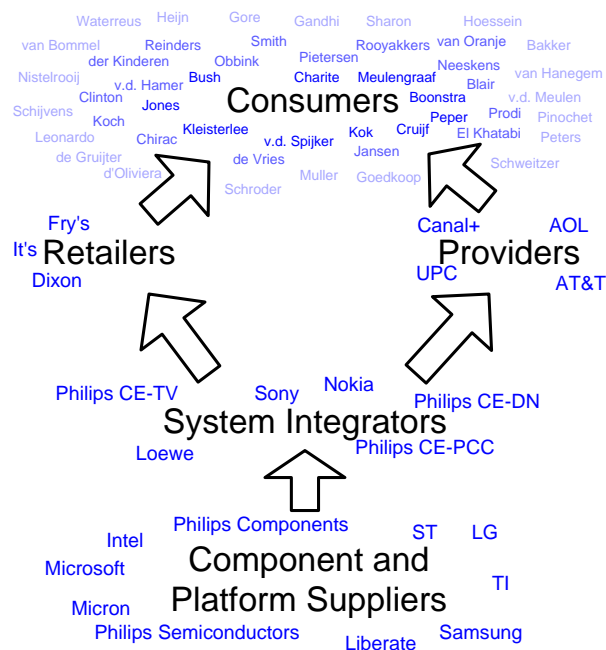


Figure 6.3: Value chain

One of the major trends in this industry is the magic buzzword *convergence*. Three more or less independent worlds of *computers*, *consumer electronics* and *telecom* are merging, see figure 6.4; functions from one domain can also be done in the other domain.

The name convergence and the visualisation in figure 6.4 suggest a more uniform set of products, a simplification. However the opposite is happening. The convergence enables integration of functions, which were separate so far for technical reasons. The technical capabilities have increased to a level, that required functionality, performance, form factor and environment together determine the products to be made. Figure 6.5 shows at the left hand side many of today's appliances, in the middle many form factors are shown and the right hand side shows some environments.

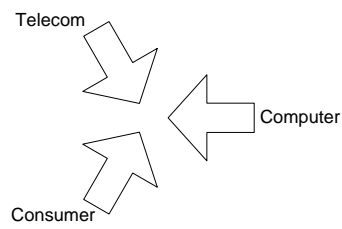


Figure 6.4: Convergence

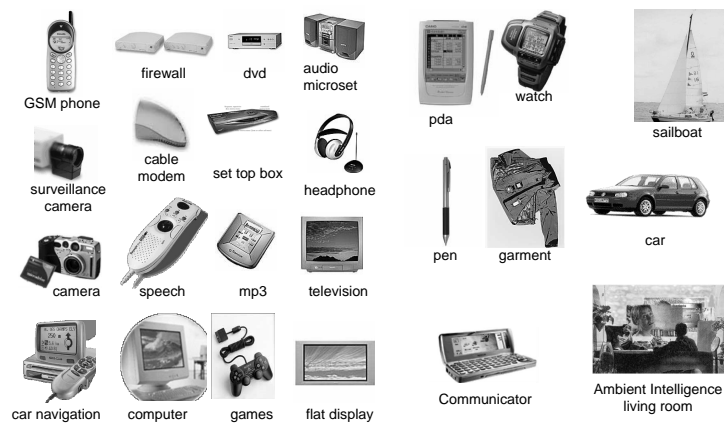


Figure 6.5: Integration and Diversity

Note that making all kind of combination products, with many different form factors for different environments and different price performance points creates a very large diversity of products!

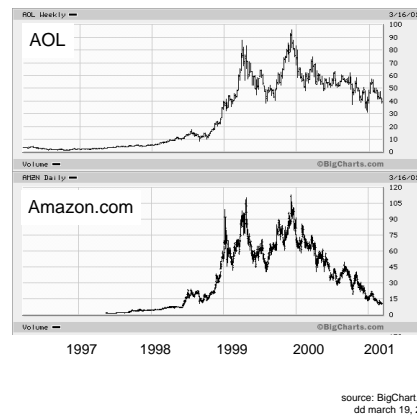


Figure 6.6: Uncertainty (Dot.Com effect)

Another market factor to take into account is the uncertainty of all players in the value chain. One of the symptoms of this uncertainty is the strong fluctuation of the stock prices, see figure 6.6.

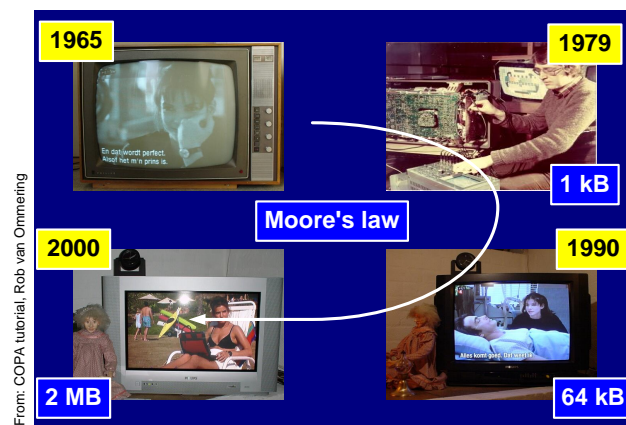


Figure 6.7: Moore's law

An historical trend is that the amount of software is increasing proportional with Moore's law, see figure 6.7.

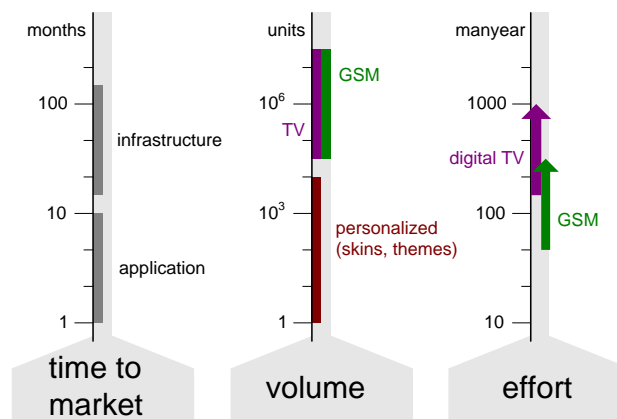


Figure 6.8: System Integrator Problem Space - Business

From business point of view the products and/or markets of the system integrators can be characterized by *time to market*, *volume*, *effort to create*. In these 3 dimensions a huge dynamic range need to be covered. Infrastructure (for instance the last mile to the home) takes a large amount of time to change, due to economical constraints, while new applications and functions need to be introduced quickly (to follow the fashion or to respond to a new killer application from the competitor). The volume is preferably large from manufacturing point of view (economy of scale), while the consumer wants to personalize, to express his identity or community (which means small scale). As mentioned before the effort to create is increasing exponentially, which means that the effort is changing order of magnitudes over decades. Figure 6.9 summarizes these characteristics.

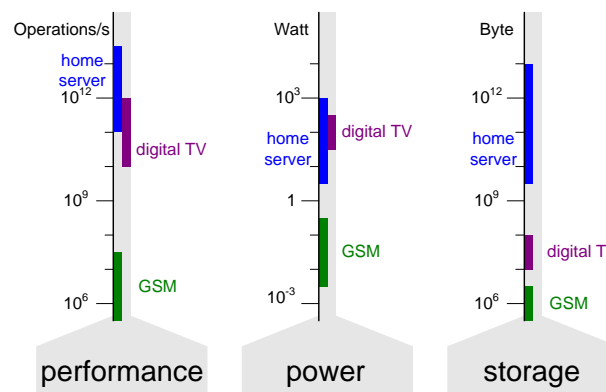


Figure 6.9: System Integrator Problem Space - Technology

Main technology concerns of the system integrators are *performance*, *power*

and *storage*. Again a huge dynamic range need to be covered in these dimensions. Video based applications have much higher processing demands than GSM speech audio. While for power portable appliances like a GSM have severe constraints and should use orders of magnitude less power than TV's or set top boxes. The amount of storage is again highly function dependent, for instance a home server which must be able to store many hours of video needs a huge amount of storage, while the address book of a GSM phone is very limited in its storage needs. The technology parameters and dynamic range are visualized in figure 6.9.

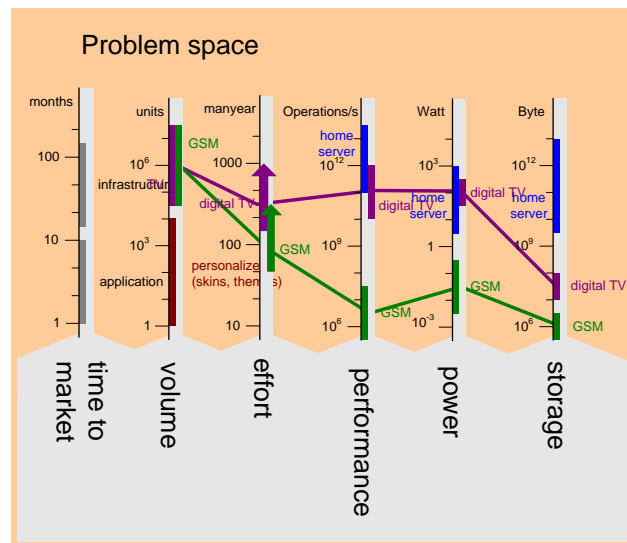


Figure 6.10: System profile

Combining the figures in one picture enables the visualization of a system profile. Figure 6.10 shows the profiles for a digital TV and for a GSM cell phone. The profile is not extended to the time to market measure, because several different time constants play a role for both GSM phones and televisions. The device itself, the applications running on the devices, the services offered on device plus infrastructure, and the infrastructure itself all have different time constants.

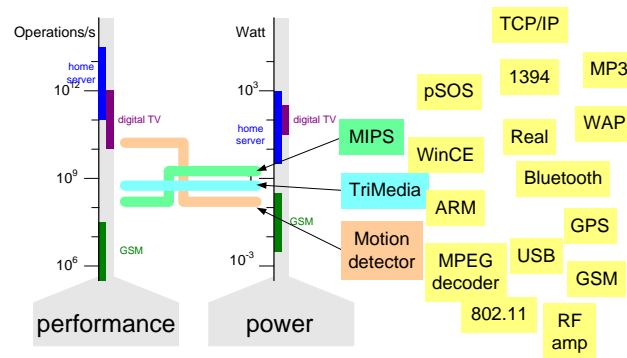


Figure 6.11: PS Technology solutions

The Philips product division Semiconductors has many hardware and software solutions available in IP-blocks. For a single problem many solutions are available. These solutions differ in their characteristics, such as *performance*, *power* and *storage*. The choice of the solution is driven by the specific product requirements. Figure 6.11 shows a subset of the available solutions and shows for 3 specific solutions their performance and power characterization.

Technologies														
	MIPS	TriMedia	MPEG decoder	ARM	Real	GSM	RF amp	Bluetooth	TCP/IP	MP3	pSOS	WinCE	1394	GPS
Systems														
watch				●	○	○	○	●	○	○	●	○		○
communicator	○	○	○	●	●	●	●	○	●	○	●	○		○
digital TV	●	●	●					○	○	○	●	○	●	
set top box	●	●	●					○	●	○	●	○	●	
pda	○	○	○	●	○	○	○	○	●	○		●		○
camcorder	●	●	●			○	○	○	○	○	●		●	○

● required  
○ optional

Figure 6.12: Partial Solution: Configurable Component Platform

The convergence problem (diversity and integration) can be tackled by a platform approach, where all the solutions must be available to be combined in one integrated solution. Figure 6.12 shows how appliances could be composed from available solutions.

Figure 6.13 summarizes the exploration of the problem and solution space. The uncertainty and diversity is addressed by *programmability*, *flexibility*, *composable architecture*, *product family approach* and *configurability*. The increased effort is addressed by *shifting development effort to suppliers*. The dynamic range of

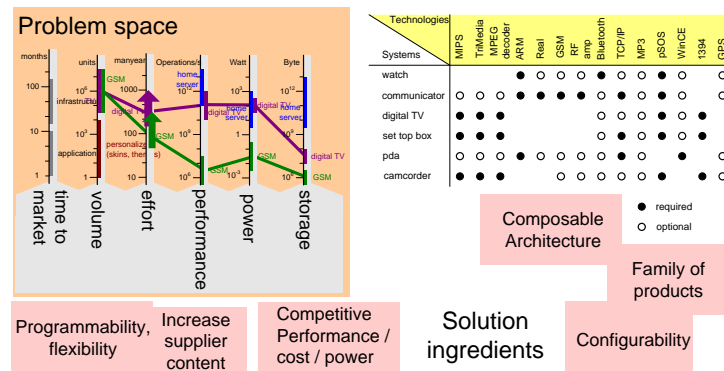
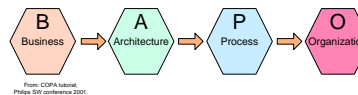


Figure 6.13: Exploring problem space and solution ingredients

requirements is addressed by supplying the right solutions at different *competitive performance/cost/power* points.



Architecture only works if the complementary viewpoints are addressed consistently

Figure 6.14: More than Architecture

This presentation focuses on architecture. Being good in architecting is not sufficient to be successful in the market. Addressing the Business, Process and Organization (People) issues also is essential for success, see figure 6.14

Figure 6.15 summarizes the conclusions of the first part of the article.



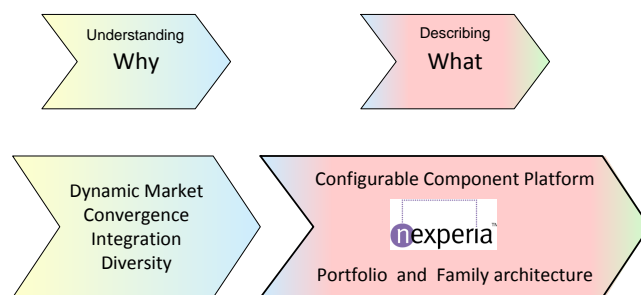


Figure 6.15: Conclusions Part 1A

## 6.3 Do the things right; Lessons from Practice

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 designers and engineers. Guiding the implementation is done by providing guidelines and high level designs for many different viewpoints. Figure 6.16 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 6.16 shows that architecting involves many more aspects and especially the integrating concepts are crucial to get working products.

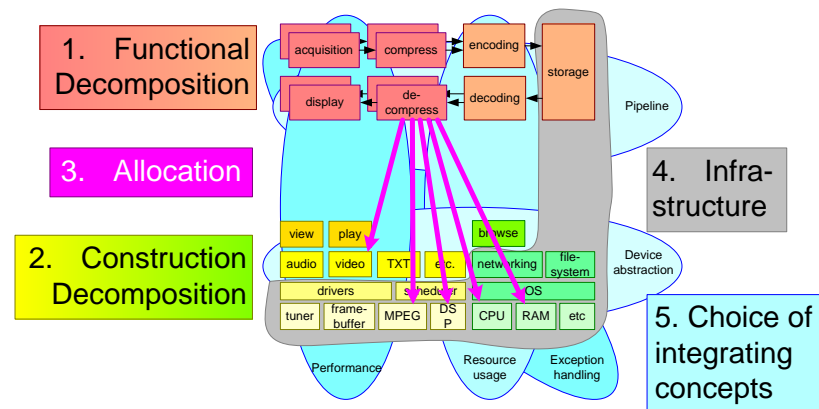


Figure 6.16: "Guiding How" by providing rules for:

Architecting involves amongst others *analyzing, assessing, balancing, making trade-offs* and *taking decisions*. This is based on architecture information and **facts**, following the needs and addressing the **expectations** of the stakeholders. A lot of the architecting is performed by the architect, which is frequently using **intuition**. As part of the architecting *vision, overview, insight* and *understanding* are created and used.

The strength of a good architect is to do this job in the real world situation, where the **facts, expectations** and intuition sometimes turn out to be false or changed! Figure 6.17 visualizes this art of architecting.

Many people expect the architect to decompose, as mentioned in the explanation of "guiding how", while integration is severely underestimated, see figure 6.18. 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.

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.

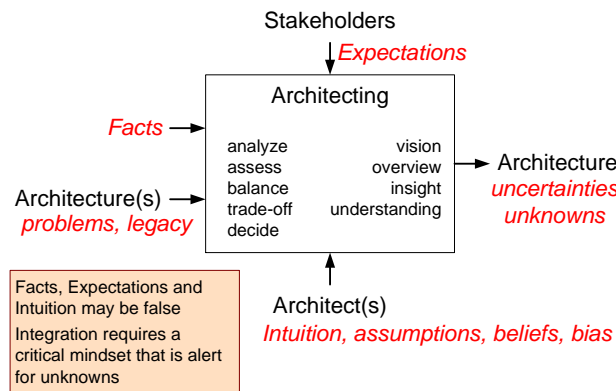


Figure 6.17: The Art of Architecting

Figure 6.19 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 served a dynamic fast changing market, see section 6.2. In other words it is a miracle if a platform is stable, when both the supplying as well as the consuming side are not stable at all.

The more academical oriented methods propose a ”first time right approach”. This sounds plausible, why waste time on wrong implementations first? The practical problem with this type of approach is that it does only work in very specific circumstances:

- well defined problem
- few people (few background, few misunderstandings)
- appropriate skill set (the so-called ”100%” instead of ”80/20” oriented people)

The first clause for our type of products is nearly always false, remember the dynamic market. The second clause is in practical cases not met (100+ manyear projects), although it might be validly pointed out that the size of the projects is the cause of many problems. The third clause is very difficult to meet, I do know only a handful of people fitting this category, none of them making out type of products (for instance professors).

Figure 6.20 shows the relationship between team size and the chance of successfully following the *first time right* approach.

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

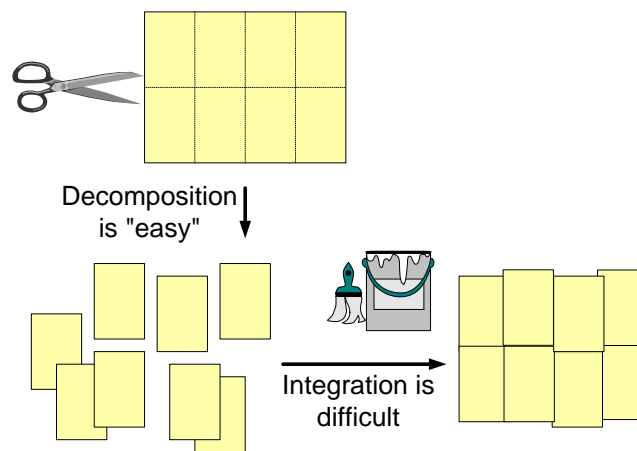


Figure 6.18: Architecting is much more than Decomposition

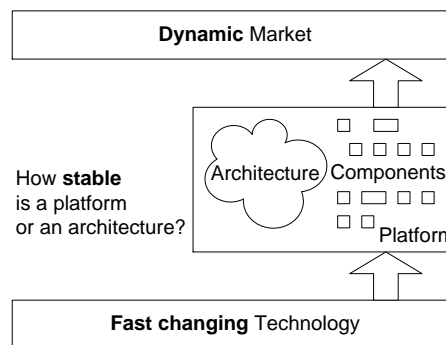


Figure 6.19: Myth: Platforms are Stable

The evolution of a platform is illustrated in figure 6.22 by showing the change in the Easyvision [16] 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.

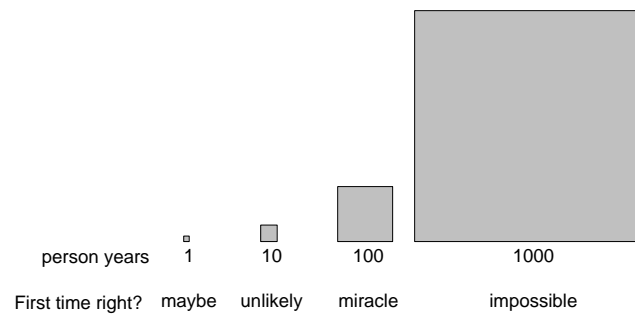


Figure 6.20: The first time right?

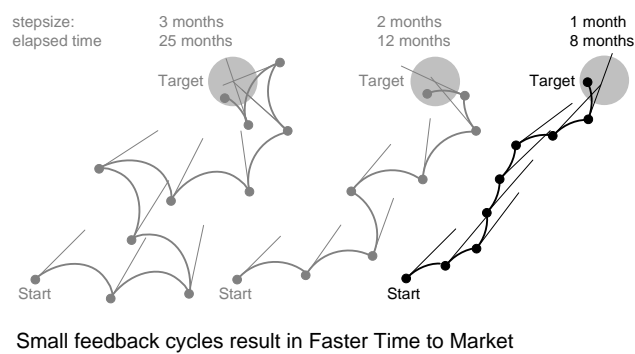


Figure 6.21: Example with different feedback cycles (3, 2, and 1 months) showing the time to market decrease with shorter feedback cycles

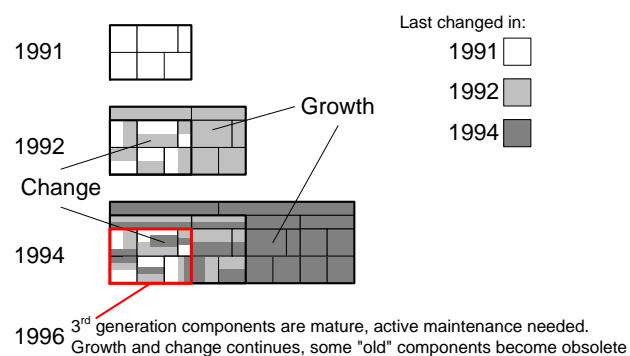


Figure 6.22: Platform Evolution (Easyvision 1991-1996)

## 6.4 The Weight of an Architecture; Architectural Chaos or Bureaucratic Control?

Does an architecture help to be successful in the business or does it harm the success? As always the answer from the architect is: it depends. The crucial success factor is the weight of the architecture.

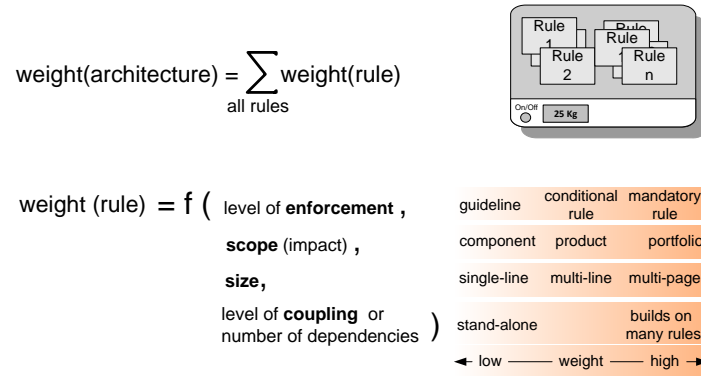


Figure 6.23: Architecture Weight

Figure 6.23 gives a definition for the weight of the architecture. The simple definition is that the overall weight of an architecture is the sum of the weight of all rules which together form the architecture.

The weight of a single rule is determined by *level of enforcement*, *scope (impact)*, *size*, *level of coupling or number of dependencies*. Figure 6.23 gives for each of these parameters a scale from low weight to high weight.

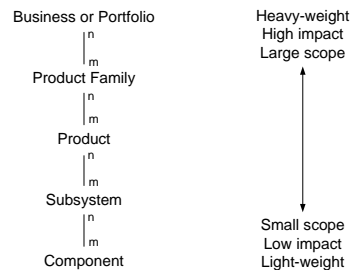


Figure 6.24: Scope and Impact

Figure 6.24 zooms in on the scope parameter to make clear the relation between the scope of the rule and the consequence for the weight.

For instance a rule like: *all the functions in all the products of the portfolio must (mandatory) return an complete predefined status object as defined in the*

system design specification, chapter exception handling, making use of the template as described in the coding standards and using the prescribed class and include files as present in the appropriate version of the code repository is a heavy rule (mandatory, portfolio scope, 4 lines of text (without the appropriate references) and depending on other entities (system design spec, template and code standard, prescribed classes and includes, repository).

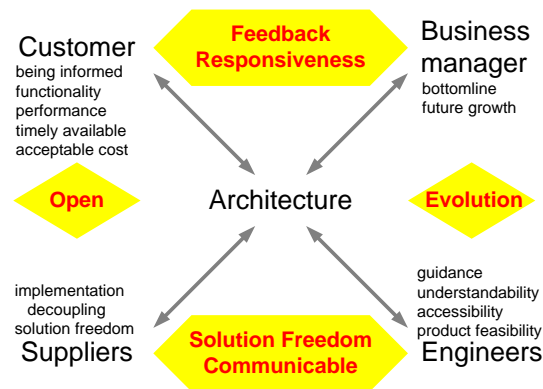


Figure 6.25: Criteria for an Architecture

So far no judgement is provided for having a *good* or *bad* architecture. In order to make a judgement we need to understand the objectives and concerns of the stakeholders. Figure 6.25 shows a number of the stakeholders and their main concerns. Note that the stakeholders have different and sometimes conflicting requirements, such is life.

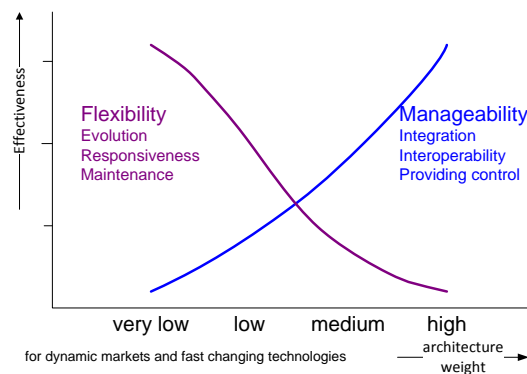


Figure 6.26: Weight versus Effectiveness

The next step in reaching a judgement is to look at the relation between effectiveness and weight. Figure 6.26 show this relationship for flexibility (evolution,

responsiveness, maintenance) and manageability (integration, interoperability, providing control). Flexibility decreases when the weight increases, while the manageability is proportional with the weight.

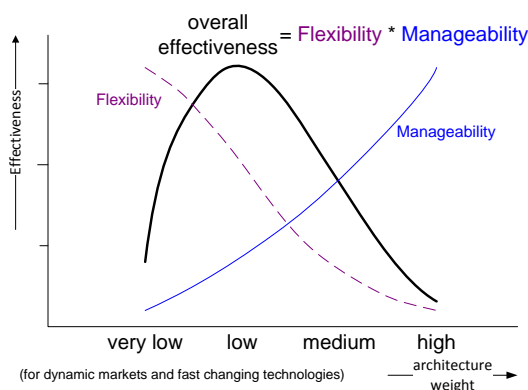


Figure 6.27: Conclusion Part 2

The question of good of bad depends on the relative importance of flexibility and manageability. For our dynamic markets and fast moving technology flexibility is very important, but for customer satisfaction the manageability is also important. The combination of these 2 requirements is shown in figure 7.17, where the curve shows that an optimum is achieved when both concerns are sufficiently met.

For different product/market/technology combinations different curves will result. Mature (stable, certain) markets with slow changing technologies, the optimum is determined by manageability and hence a heavy architecture. At the other extreme very dynamic markets and technologies, with forgiving customers or low economic risks will benefit from extreme flexibility and a very light architecture.



## 6.5 Light weight how-to

With both the definition of the weight of the architecture and the insight that this weight should be low it becomes possible to look for ways to minimize the weight.

An high impact way of minimizing the weight is to minimize the number of the rules. This can be achieved by providing as much as focus to the architecture as possible. Focus is derived from the market and customer needs. Understanding of the customer helps to understand and therefore to sharpen the requirements, see figure 6.28.

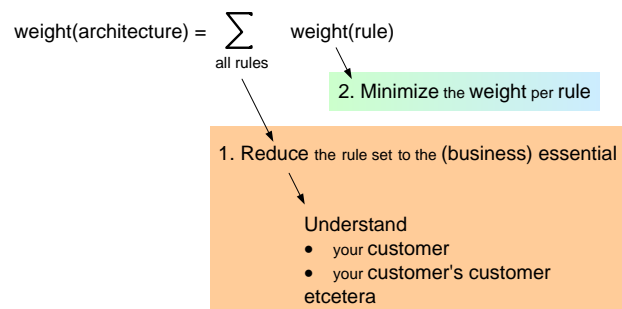


Figure 6.28: Light Weight How -To

The next step is to minimize the weight of every individual rule. Every parameter influencing the weight of a rule must be minimalized. The *level of enforcement* can be minimized by making as few as possible rules mandatory, work with guidelines as much as possible. The *scope* can be minimized by empowering and delegating as much as possible, in other words let component or subsystem architects make local rules (or better guidelines) for their specific scope. The *size* of a rule is minimized by leaving out details in the rule itself, short conceptual rules are very powerful. The *level of coupling* is minimized by "designing" the architecture rules. Especially multi-view architecting helps to cope with the highly complex reality. One dimensional decompositions result in highly coupled rules to capture aspects from other dimensions.

Figure 6.29 visualizes the way to minimize the individual weight of a rule.

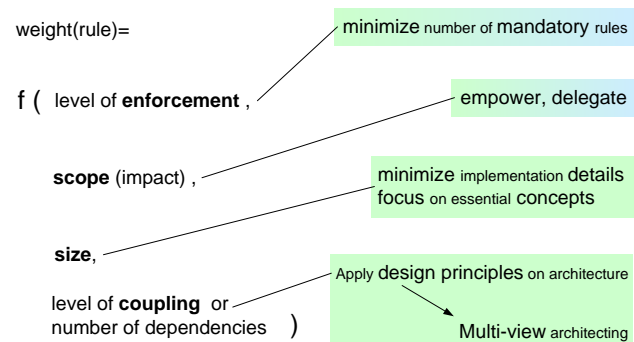


Figure 6.29: Minimize Rule Weight

## 6.6 Summary

Figure 6.30 summarizes the full presentation. The market to be served is highly dynamic. Lessons from practice show that changes are normal, stability of the solution is the exception. In this dynamic market with a changing solution space the architecture must be light weight.

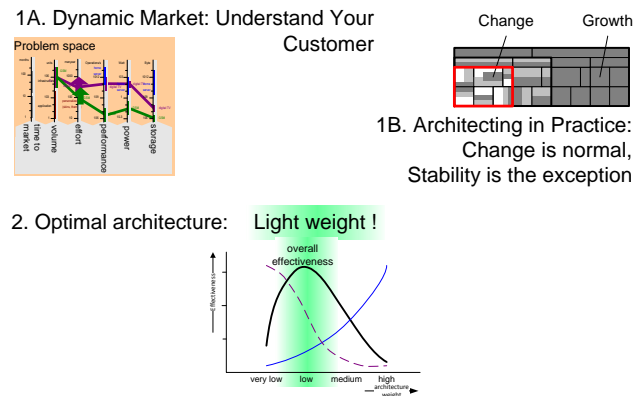


Figure 6.30: Summary

## 6.7 Acknowledgements

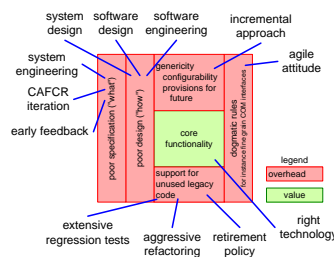
This presentation has been enabled by the inspiring and critical comments of:

- Jürgen Müller
- Peter van den Hamer
- Lex Heerink
- William van der Sterren

Remarks and textual improvements were made by Pierre van de Laar.

## Chapter 7

# Exploration of the bloating of software



### 7.1 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 7.2 and 7.3.

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

### 7.2 Module level bloating

Figure 7.1 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

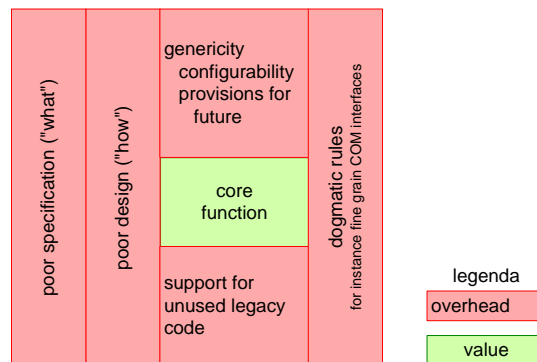


Figure 7.1: Exploring bloating

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).

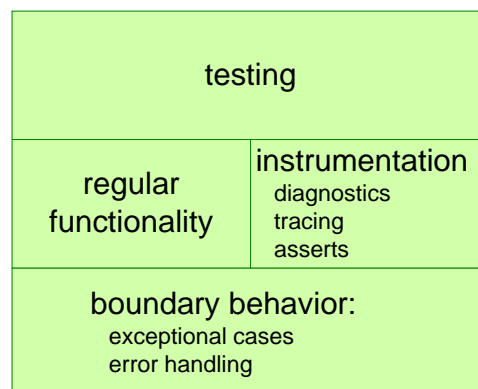
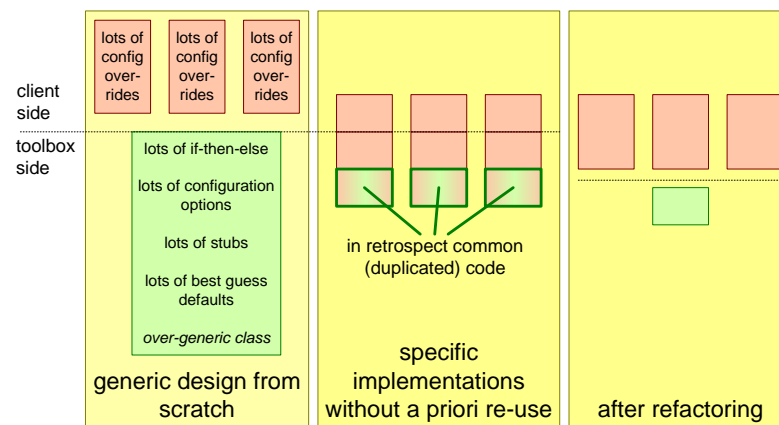


Figure 7.2: Necessary functionality is more than the intended regular function

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 7.2. 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 7.3 shows an actual example of part of the Medical Imaging system [16], 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 7.3: 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.

### 7.3 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 7.4 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.

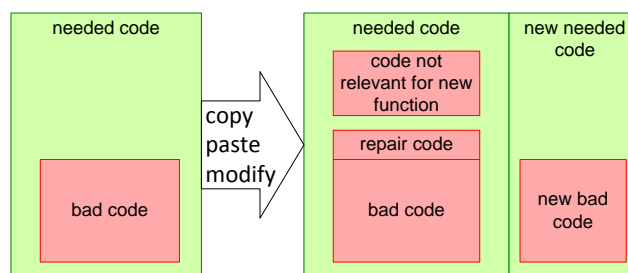


Figure 7.4: Shit propagation via copy paste

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

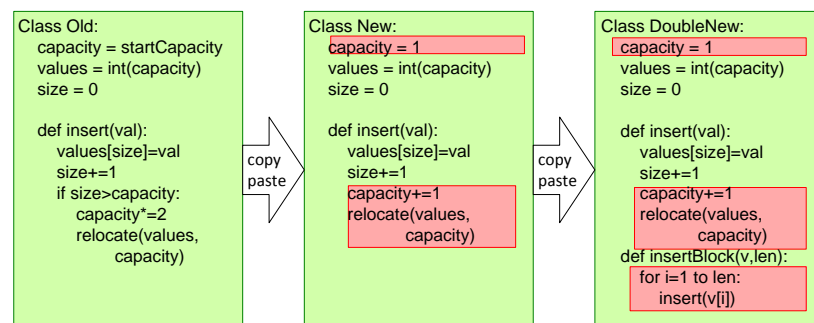


Figure 7.5: Example of shit propagation

An example of such shit propagation is shown in figure 7.5. 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.

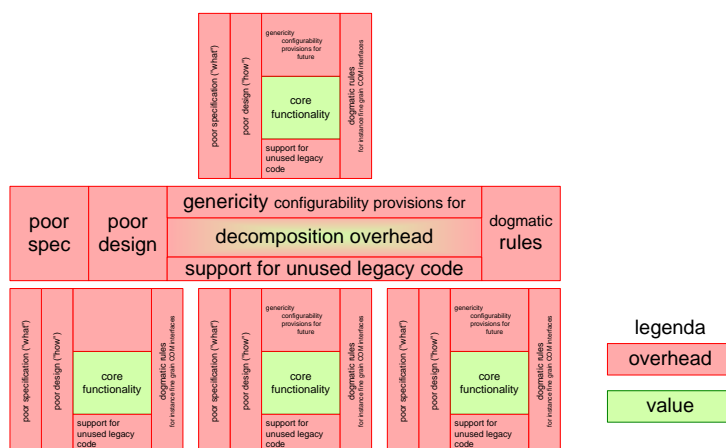


Figure 7.6: Bloating causes more bloating

One of the bloating problems is that bloating causes more bloating, as shown in figure 7.6. 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 7.7.

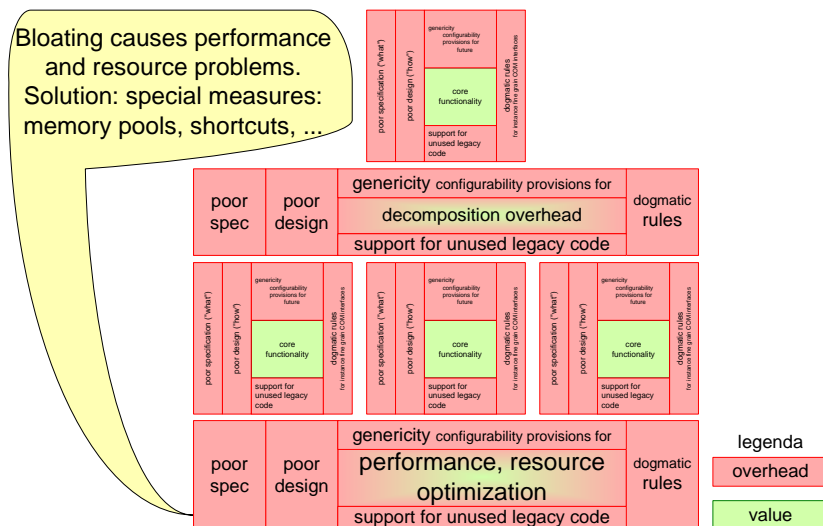


Figure 7.7: causes even more bloating...

## 7.4 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 7.8 shows some of the consequences.

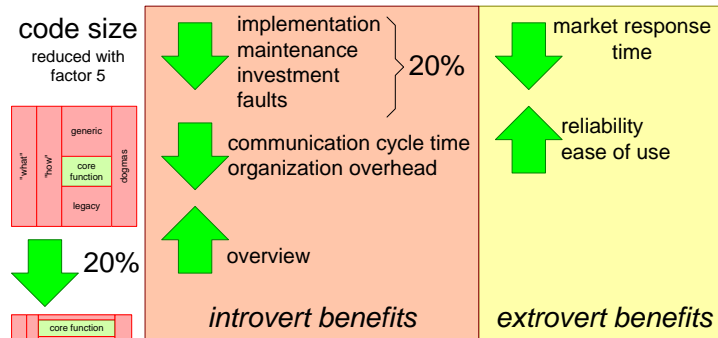


Figure 7.8: What if we remove half of the bloating?

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 7.9 shows the relation between crew size, organization and housing.

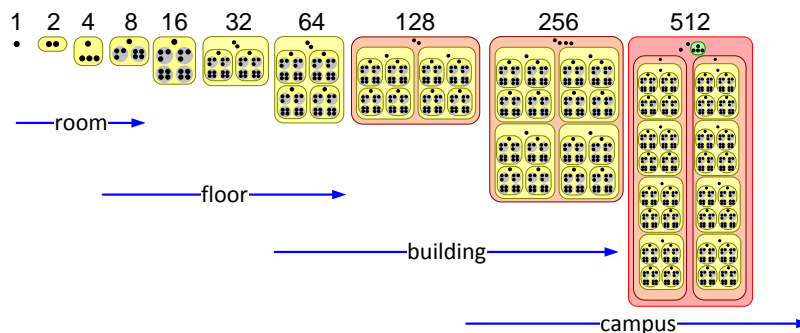


Figure 7.9: Impact of size on organization, location, process

Of course a reduction with a factor 5 is a tremendous challenge, a plan to attack the bloating is discussed in section 7.5. 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.

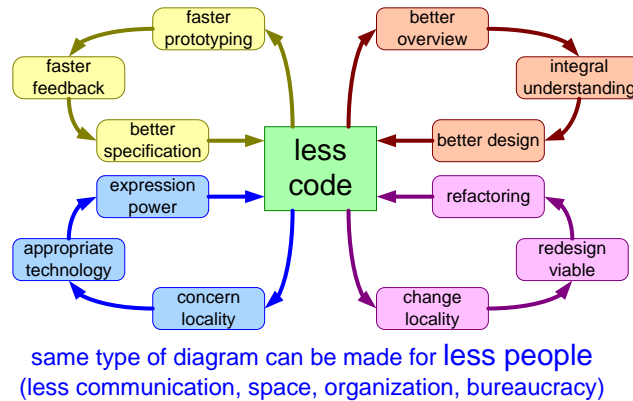


Figure 7.10: Anti bloating multiplier

If we are able to reverse the trend of bloating, an anti bloating multiplier effect will help us, as shown in figure 7.10. 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!

## 7.5 How to attack the bloating?

The bloating must be attacked by coping with all the different causes of bloating as discussed in section 7.2. Figure 7.11 summarizes all different approaches that can be used to attack these different causes.

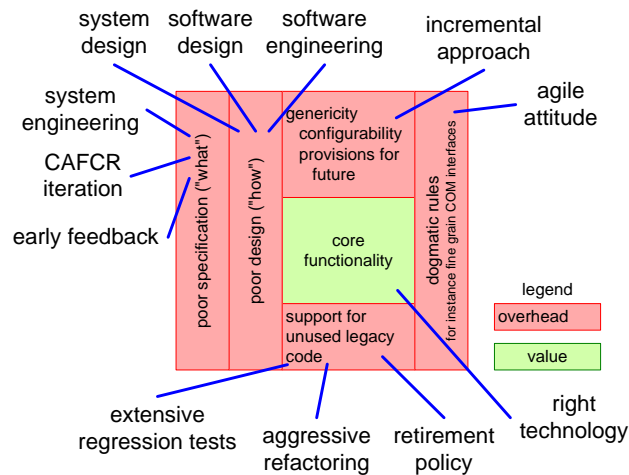


Figure 7.11: How to reduce bloating

### 7.5.1 Improving the specification

The systems engineering discipline is a matured discipline, for instance in the military and aero space domain, see for instance: [8] and [11]. 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 [6], [4] and [10].

In [21] 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.

### 7.5.2 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

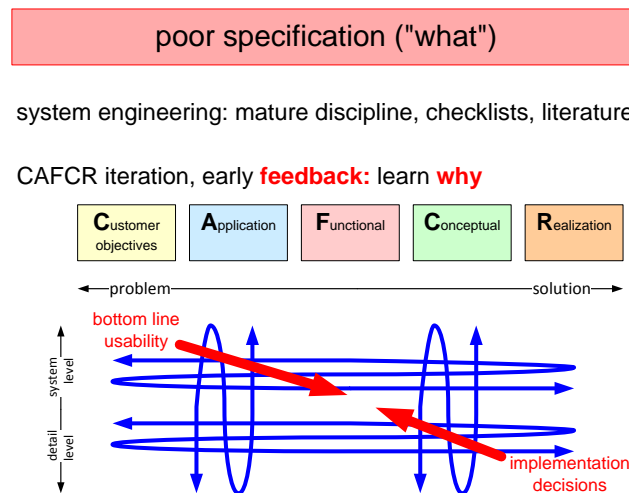


Figure 7.12: Improving the specification

account. The architectural reasoning method as described in [21] emphasizes the need for multiple views and methods in order to cope with many relevant design dimensions.

Figure 7.13 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.

### 7.5.3 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

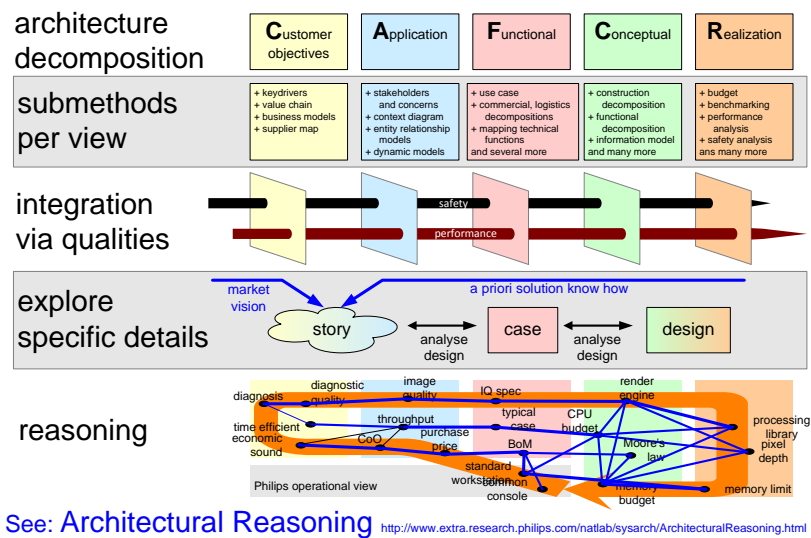


Figure 7.13: Use multiple views and methods

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 7.14 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 7.15

#### 7.5.4 Match solution technology with problem

The size of the functionality itself can often be reduced by using the appropriate technology for the specific type of problem. Figure 7.16 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

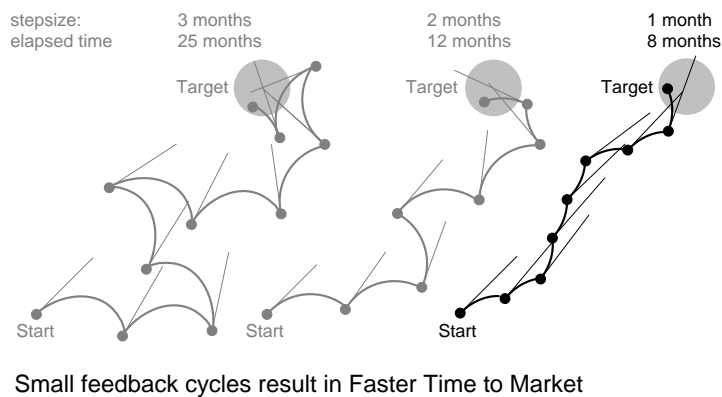


Figure 7.14: Feedback (3)

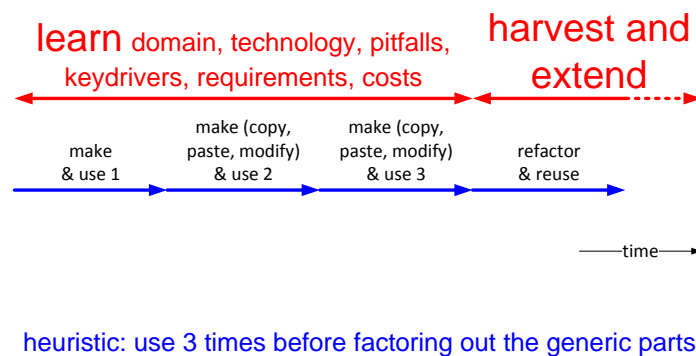


Figure 7.15: Lesson learned about reuse

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.

### 7.5.5 Agility instead of dogmatism

An agile attitude is needed to avoid dogmatic application of all kinds of architecture rules. In [19] recommendations are given to achieve a light weight architecture.

Figure 7.17 from [19] 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 increase of *manageability*, which can backfire if this translates in dogmatism.




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 7.16: Examples of "right" technology choices

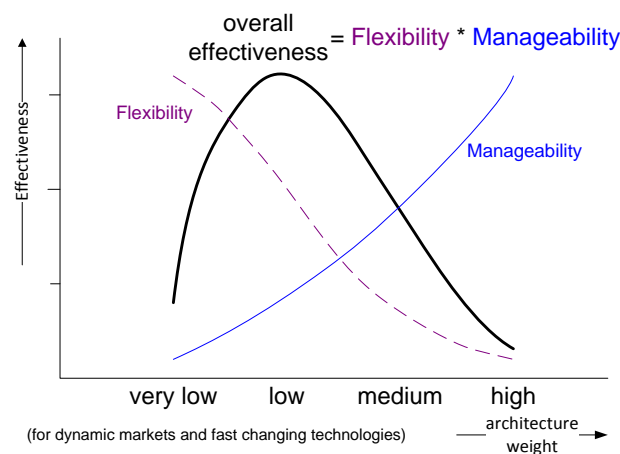


Figure 7.17: Keep the architecture weight low

### 7.5.6 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 fear of cleanup.

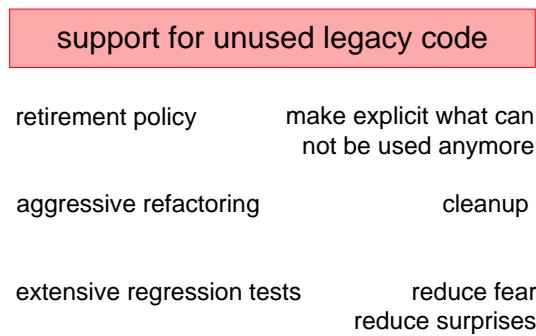


Figure 7.18: Reduce unused code

## 7.6 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.

# Bibliography

- [1] Chris M.S. Abts, Barry W. Boehm, and Elizabeth Bailey Clark. COCOTS: A COTS software integration lifecycle cost model-model overview and preliminary data collection findings. <http://sunset.usc.edu/publications/TECHRPTS/2000/usccse2000-501/usccse2000-501.pdf>, 2000.
- [2] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, MA, 2000.
- [3] Barry W. Boehm et al. Cocomo ii homepage. <http://sunset.usc.edu/research/COCOMOII/index.html>, 2000.
- [4] B.W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, May 1988.
- [5] Dana Bredemeyer. Definitions of software architecture. <http://www.bredemeyer.com/definiti.htm>, 2002. large collection of definitions of software architecture.
- [6] Thomas Gilb. Evolutionary object management. <http://www.gilb.com/Download/EVOART95.ZIP>, 1996.
- [7] Derek K. Hitchins. Putting systems to work. <http://www.hitchins.co.uk/>, 1992. Originally published by John Wiley and Sons, Chichester, UK, in 1992.
- [8] INCOSE. International council on systems engineering. <http://www.incose.org/toc.html>, 1999. INCOSE publishes many interesting articles about systems engineering.
- [9] Carnegie Mellon Software Engineering Institute. How do you define software architecture? <http://www.sei.cmu.edu/architecture/definitions.html>, 2002. large collection of definitions of software architecture.

- [10] Philippe B. Kruchten. A rational development process. *Crosstalk* 9, pages 11–16, July 1996.
- [11] James N. Martin. *Systems Engineering Guidebook*. CRC Press, Boca Raton, Florida, 1996.
- [12] Gerrit Muller. Granularity of documentation. <http://www.gaudisite.nl/DocumentationGranularityPaper.pdf>, 1999.
- [13] Gerrit Muller. Product families and generic aspects. <http://www.gaudisite.nl/GenericDevelopmentsPaper.pdf>, 1999.
- [14] Gerrit Muller. Requirements capturing by the system architect. <http://www.gaudisite.nl/RequirementsPaper.pdf>, 1999.
- [15] Gerrit Muller. The system architecture homepage. <http://www.gaudisite.nl/index.html>, 1999.
- [16] Gerrit Muller. Case study: Medical imaging; from toolbox to product to platform. <http://www.gaudisite.nl/MedicalImagingPaper.pdf>, 2000.
- [17] Gerrit Muller. Process decomposition of a business. <http://www.gaudisite.nl/ProcessDecompositionOfBusinessPaper.pdf>, 2000.
- [18] Gerrit Muller. From legacy to state-of-the-art; architectural refactoring. <http://www.gaudisite.nl/ArchitecturalRefactoringPaper.pdf>, 2001.
- [19] Gerrit Muller. Light weight architectures; the way of the future? <http://www.gaudisite.nl/info/LightWeightArchitecting.info.html>, 2001.
- [20] Gerrit Muller. The system architect; meddler or savior? <http://www.gaudisite.nl/MeddlerOrSaviorPaper.pdf>, 2001.
- [21] Gerrit Muller. Architectural reasoning explained. <http://www.gaudisite.nl/ArchitecturalReasoningBook.pdf>, 2002.
- [22] Gerrit Muller. The importance of system architecting for development. <http://www.gaudisite.nl/ImportanceOfSAforDevelopmentPaper.pdf>, 2002.
- [23] Gerrit Muller. CAFCR: A multi-view method for embedded systems architecting: Balancing genericity and specificity. <http://www.gaudisite.nl/ThesisBook.pdf>, 2004.

- [24] Gerrit Muller. Key drivers how to. <http://www.gaudisite.nl/KeyDriversHowToPaper.pdf>, 2010.
- [25] Gerrit Muller. Requirements elicitation and selection. <http://www.gaudisite.nl/RequirementsElicitationAndSelectionPaper.pdf>, 2010.
- [26] Henk Obbink, Jürgen Müller, Pierre America, and Rob van Ommering. COPA: A component-oriented platform architecting method for families of software-intensive electronic products. [http://www.hitech-projects.com/SAE/COPA/COPA\\_Tutorial.pdf](http://www.hitech-projects.com/SAE/COPA/COPA_Tutorial.pdf), 2000.

## History

**Version: 0.4, date: July 13, 2010 changed by: Gerrit Muller**

- Added "A Method to Explore Synergy between Products"

**Version: 0.3, date: June 6, 2003 changed by: Gerrit Muller**

- Added "How to Create a Manageable Platform Architecture?"

**Version: 0.2, date: June 6, 2003 changed by: Gerrit Muller**

- Added "Exploration of the bloating of software"

**Version: 0.1, date: March 7, 2003 changed by: Gerrit Muller**

- Added "Software Reuse; Caught between strategic importance and practical feasibility" no changelog yet

**Version: 0, date: June 13, 2002 changed by: Gerrit Muller**

- Created very preliminary bookstructure, no changelog yet