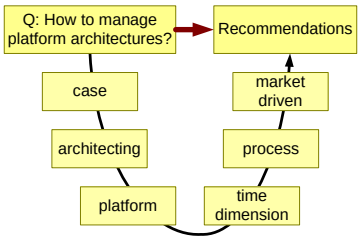
How to Create a Manageable Platform Architecture?

-



Gerrit Muller
University of South-Eastern Norway-NISE
Hasbergsvei 36 P.O. Box 235, NO-3603 Kongsberg Norway
gaudisite@gmail.com

Abstract

Today's fast pace of the market and the technology development forces the product creators to rethink their development approach. One of the directions is to maximize the return on investments of frequently used functions, for instance by re-use, component based design or by a platform approach. The architecting effort is a key success factor to combine re-use approaches with fast and innovative product creation.
In this presentation we will present a case, discuss the role of the architecture, and elaborate the essential architecture ingredients for a successful platform creation, and evolution, and innovative product creation.

version: 1.0 status: concept September 6, 2020

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.

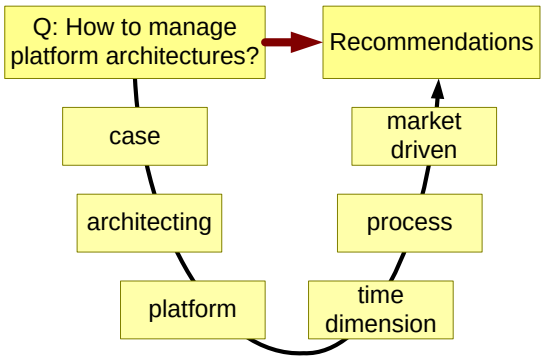


Figure 1: Outline of this paper

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 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.

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.

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 2.

The common technology in conventional X-ray systems is developed by component oriented business groups, which make generators, tubes, camera's, detectors, et

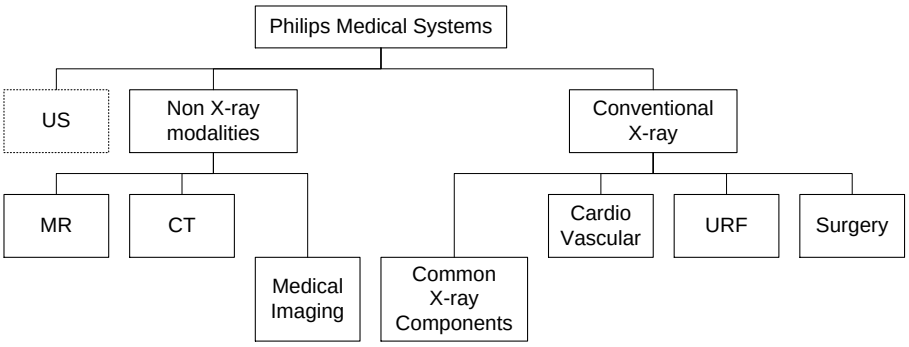


Figure 2: Philips Medical Systems, schematic organization overview.

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[1] 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 3 shows a generic set of Radiology drivers.

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

2.2 Historic Phases

The development model of Medical Imaging has changed several times. Roughly the phases in Figure 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.

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

[1] 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.

[2] 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.

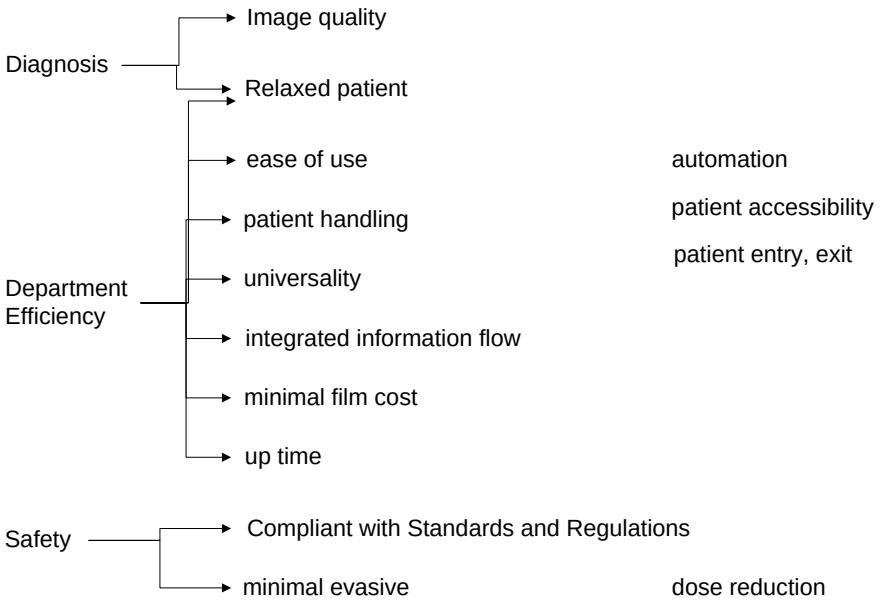


Figure 3: Generic drivers of Radiology Departments

be possible when a common implementation is used. The lessons learned during this long struggle have been partially consolidated in [4].

The group of people, which started the Common Viewing development, applied a massive amount of technology innovations, see Figure 5.

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

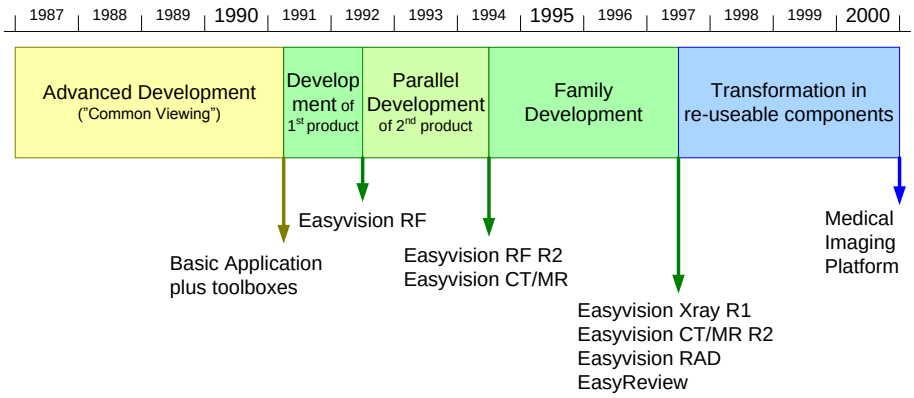


Figure 4: Phases of Medical Imaging

2.4 Medical Imaging X-Ray

Figure 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.

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.

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 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

- 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 5: Technology innovations by Common Viewing

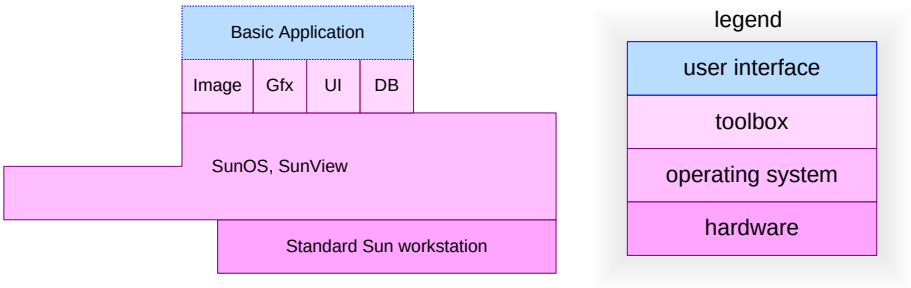


Figure 6: Idealized layering of SW toolboxes and Basic Application in september 1991

Medical Imaging was not sold as a disgusting computer or workstations, rather it was positioned as a clinical appliance.

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 10 indicate new SW. The amount of code increased from 100 klines to 350 klines of code.

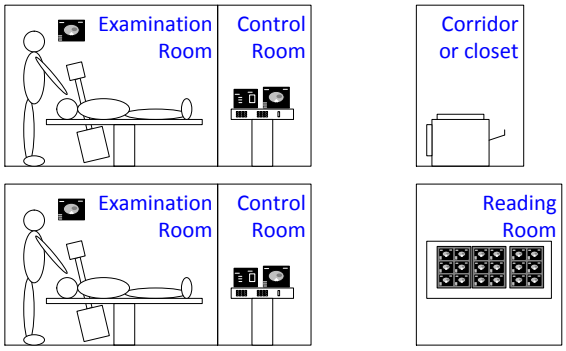


Figure 7: X-ray rooms from examination to reading around 1990

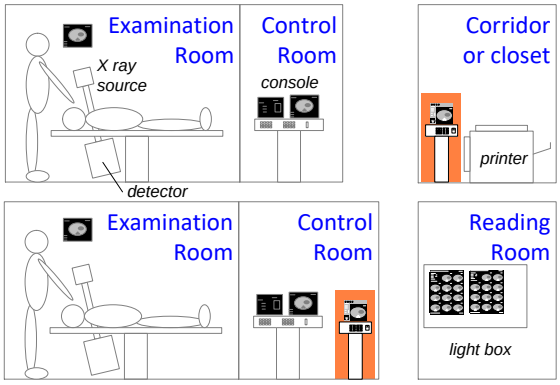


Figure 8: X-ray rooms from examination to reading, when Medical Imaging is applied as printserver

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.

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 Reformatting (MPR) functionality, where arbitrary slices are reconstructed from the volume data set.

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

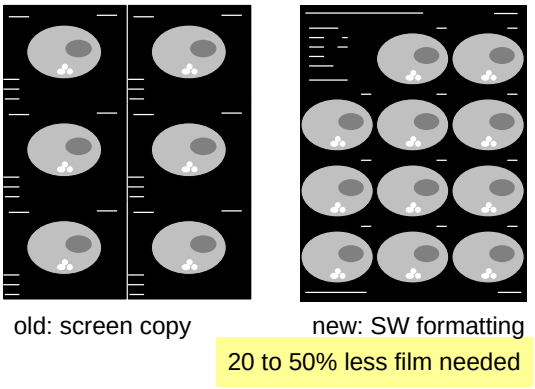


Figure 9: Comparison of conventional *screen copy* based film and a film produced by Medical Imaging. This case is very favorable for the Medical Imaging approach, typical gain is 20% to 50%.

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 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.

All diagrams 6, 10 and 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.

2.6 Towards Workflow

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

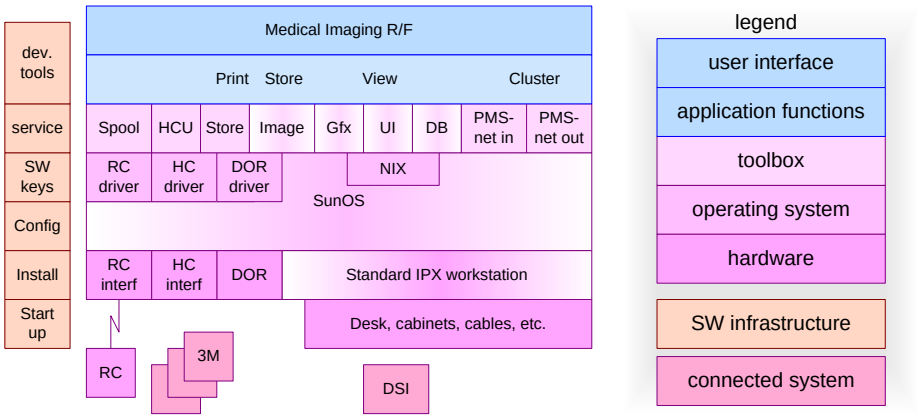


Figure 10: Idealized layers of the Medical Imaging R/F software in september 1992

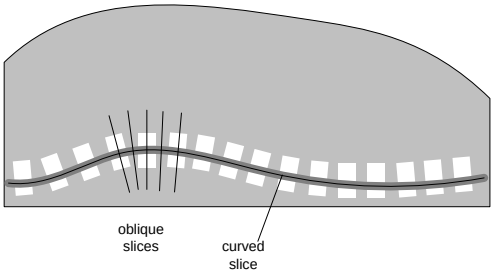


Figure 11: Example of Multi Planar Reformatting applied on the spine

broader context of radiology and even the diagnostic workflow. Figures 14 and 15 show the increasing context where the workstation technology can be deployed.

The increasing context causes new extensions of the SW building, as shown in Figure 16.

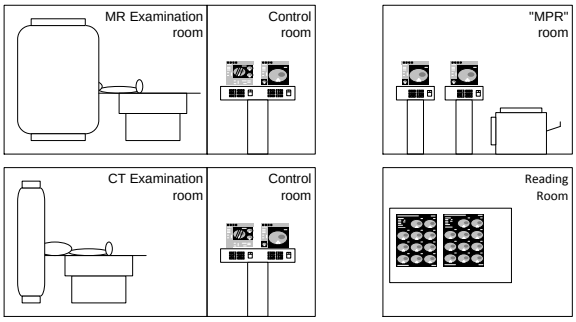


Figure 12: Example of CT and MR department, where Medical Imaging is deployed

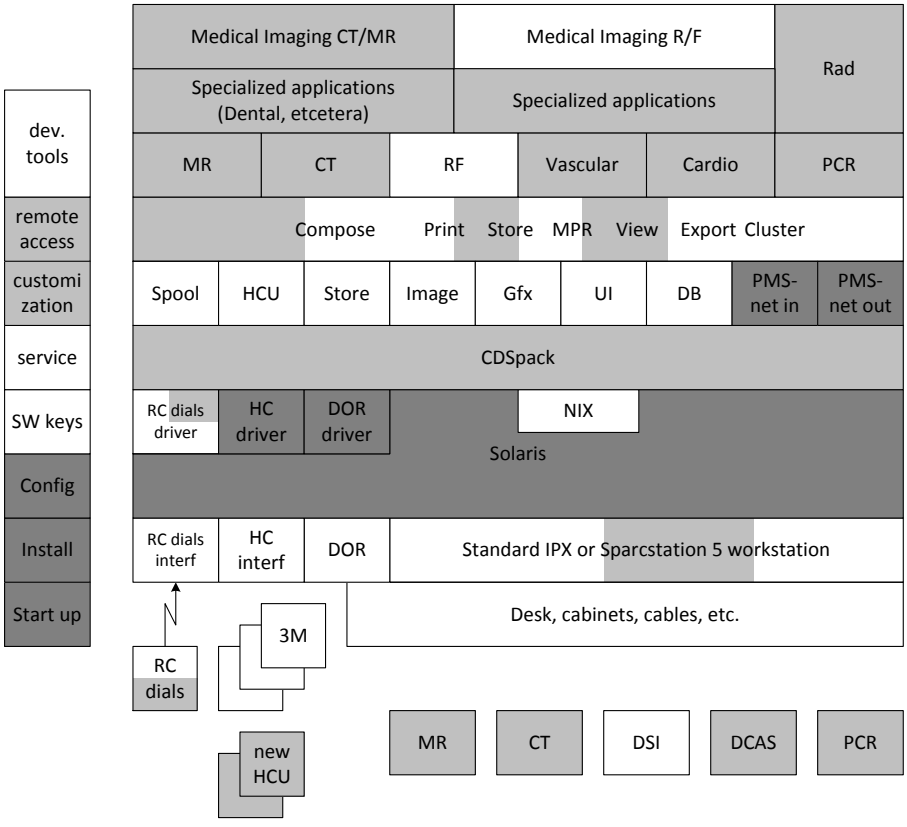


Figure 13: Idealized layers of the Medical Imaging software in june 1994

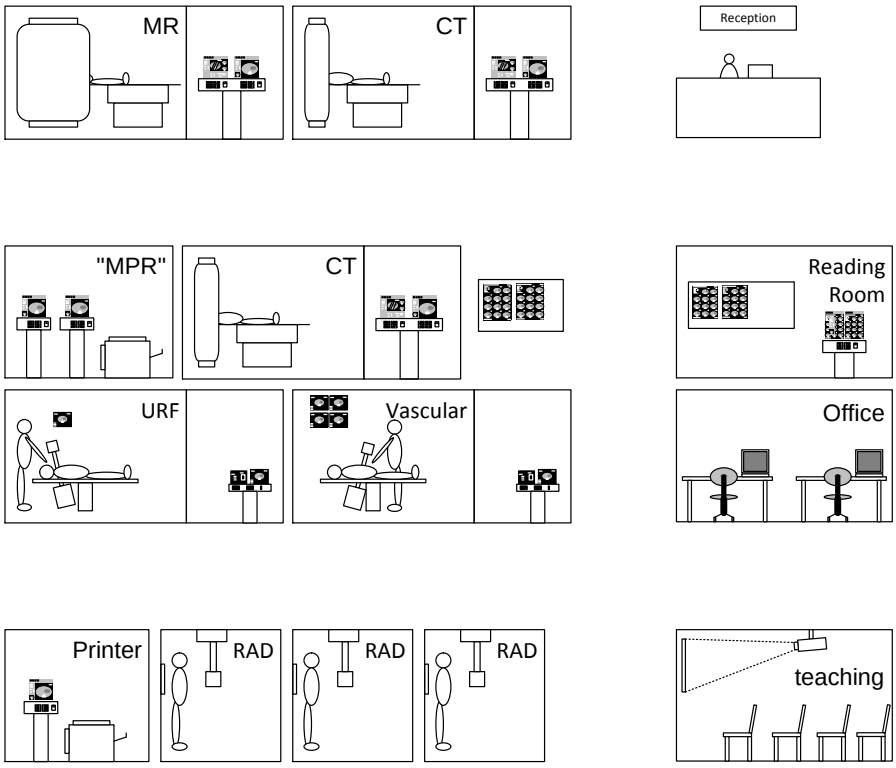


Figure 14: Radiology department as envisioned in 1996

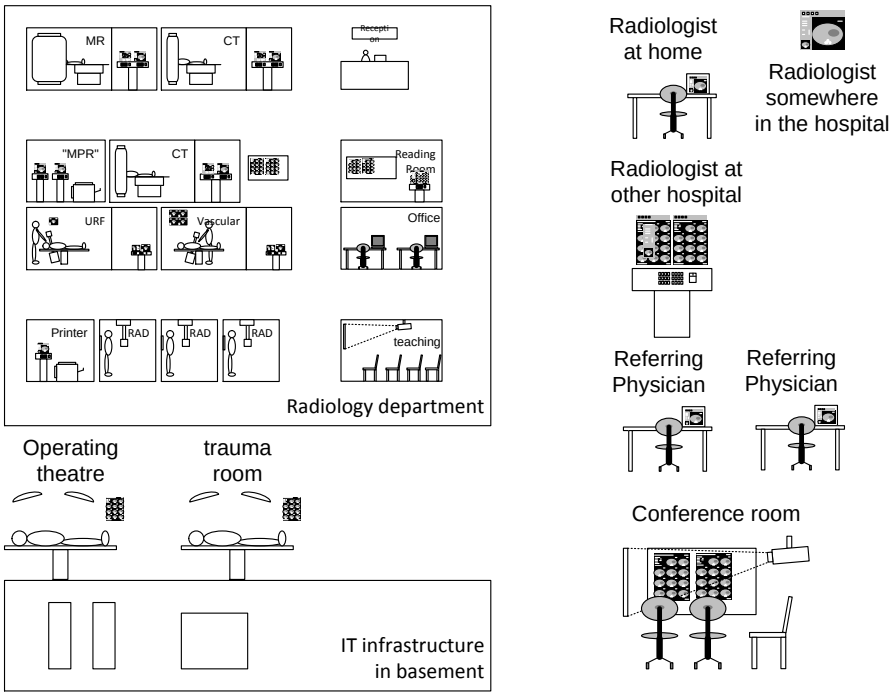


Figure 15: Medical Imaging in health-care workflow perspective, as envisioned in 1996

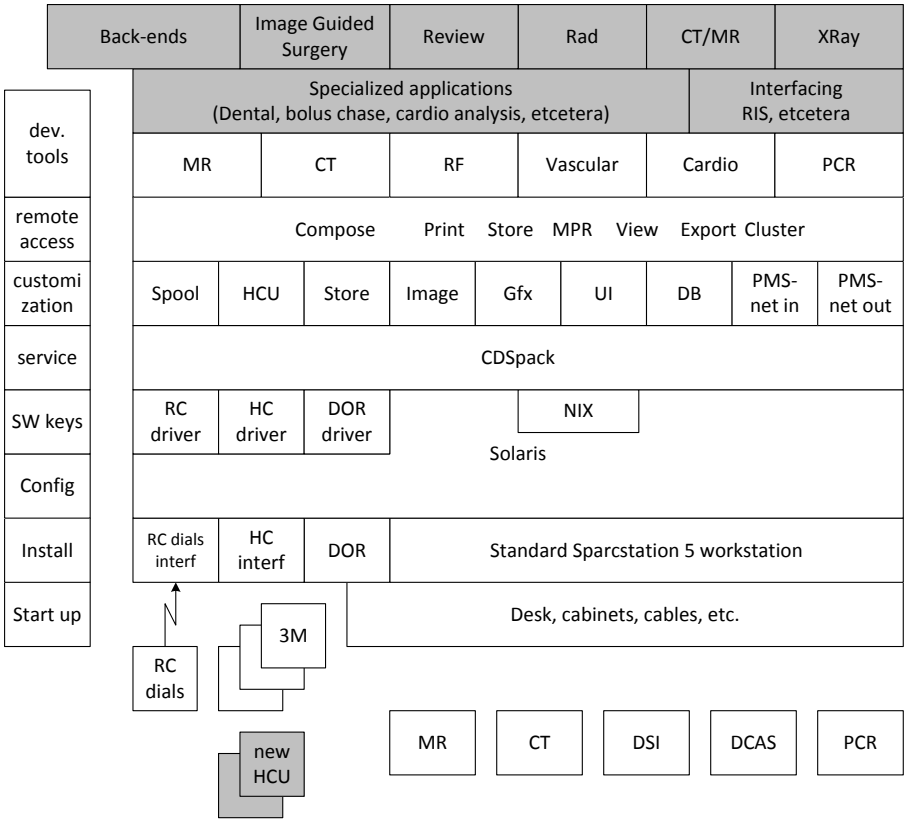


Figure 16: Idealized layers of the Medical Imaging software in 1996

3 Architecture

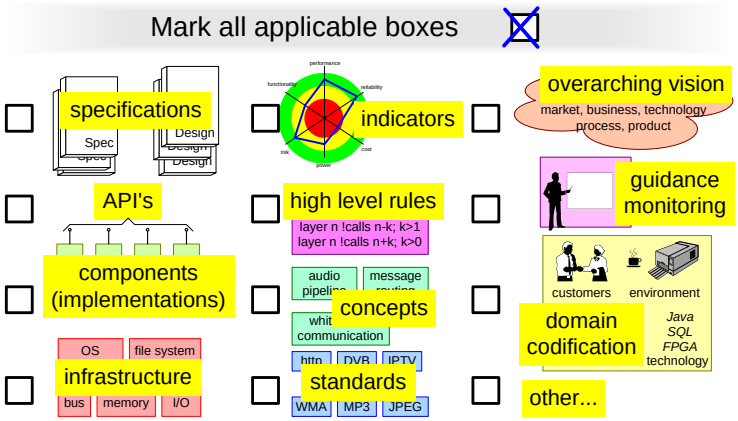


Figure 17: What is Architecture?

What is *Architecture*? Every individual appears to use their own definition of architecture. Figure 17 shows many different aspects that are frequently mentioned as being part of the architecture.

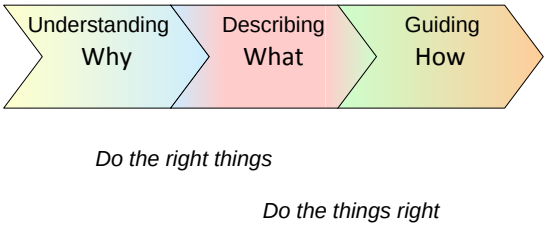


Figure 18: What is Architecture?

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 18.

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 19 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 19

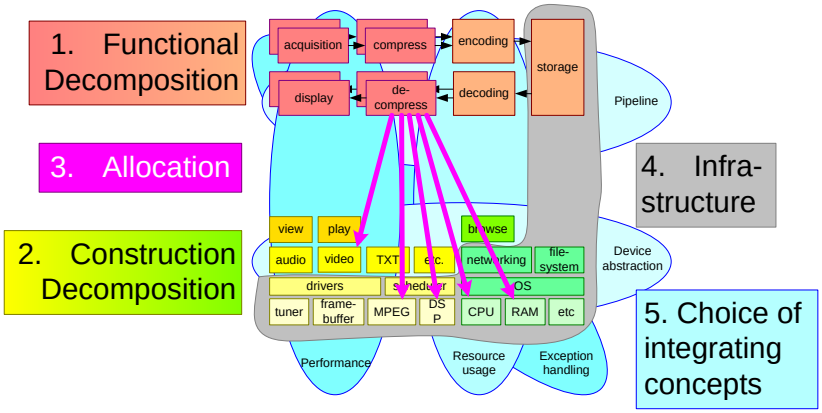


Figure 19: ”Guiding How” by providing rules for:

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.

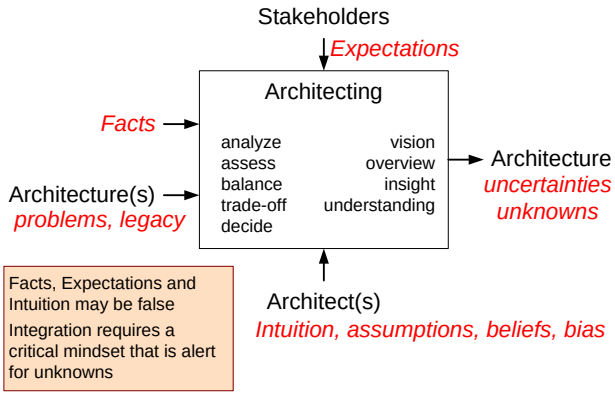


Figure 20: The Art of Architecting

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 top be false or changed! Figure 20 visualizes this art of architecting.

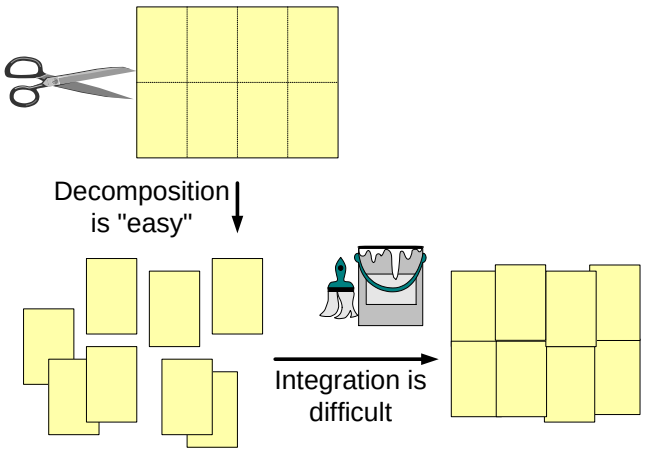


Figure 21: Architecting is much more than Decomposition

Many people expect the architect to decompose, as mentioned in the explanation of "guiding how", while integration is severely underestimated, see figure 21. 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.

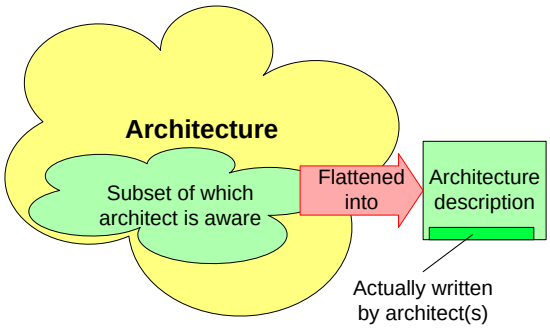


Figure 22: The architecture description is by definition a flattened and poor representation of an actual architecture.

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[3].

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

[3]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 [2], [1], or [3]

in figure 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.

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.

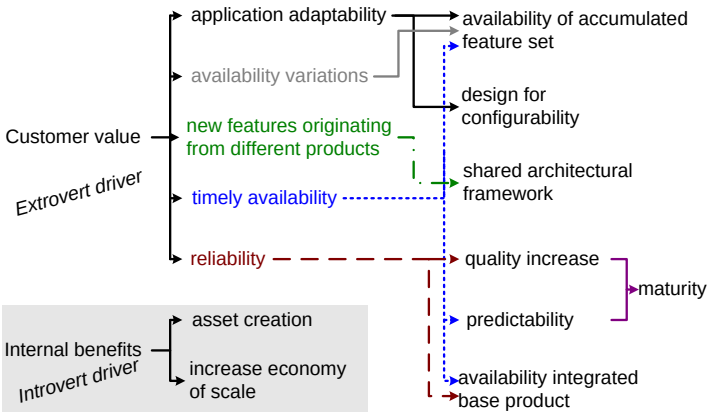


Figure 23: Drivers of Generic Developments

Figure 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”

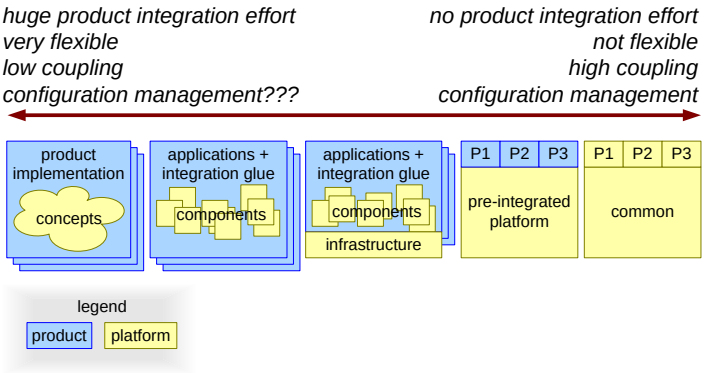


Figure 24: What is a Platform?

But what is a *platform*? Many different types of platforms can be found. Figure 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 it’s 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 25 shows these deliverables, and Figure 26 shows the same

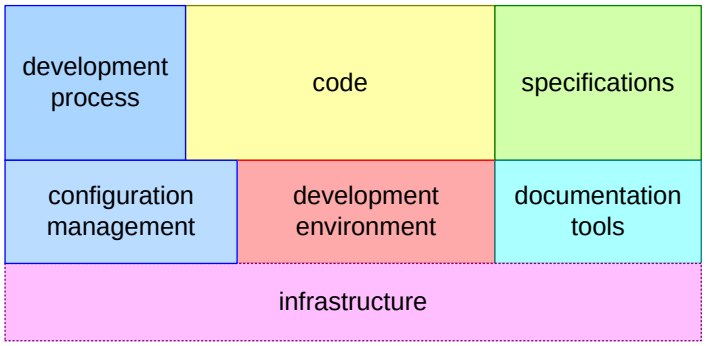


Figure 25: Platform Source Deliverables

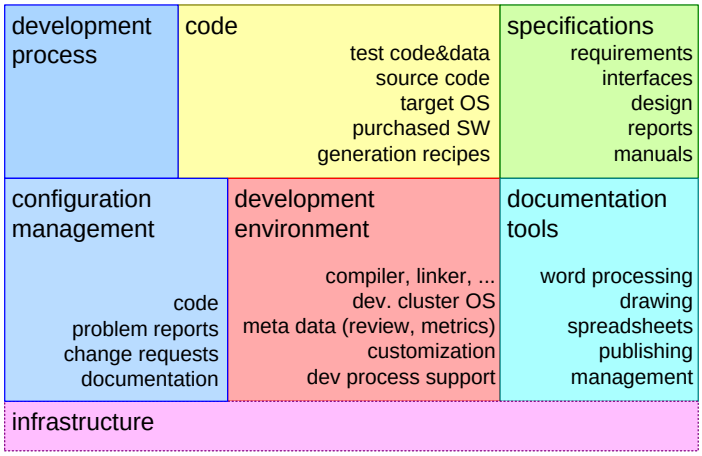


Figure 26: And now in More Detail...

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 2, used a platform approach to share common functions. In the table in Figure 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
value metric	applications	1	4	8	16	32
	number of inputs (a.o. modalities)	1	5	10	15	
number of people	platform			35	37	38
	applications			27	35	41
	total		52	62	72	79
efficiency	people per application		13	8	5	3

Figure 27: Example of Platform Efficiency

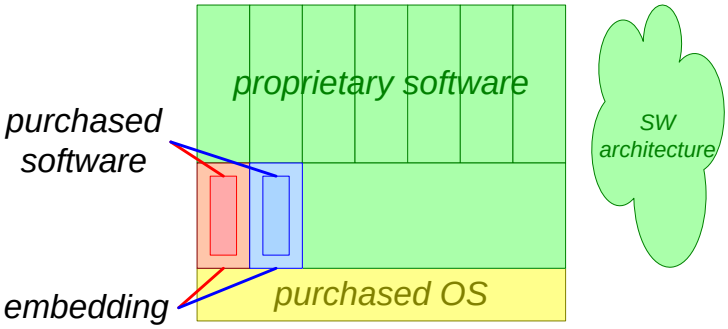


Figure 28: Purchased SW Requires Embedding

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 28 shows that purchased software requires some kind of embedding to fit it into the desired architecture.

Figure 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 30 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.

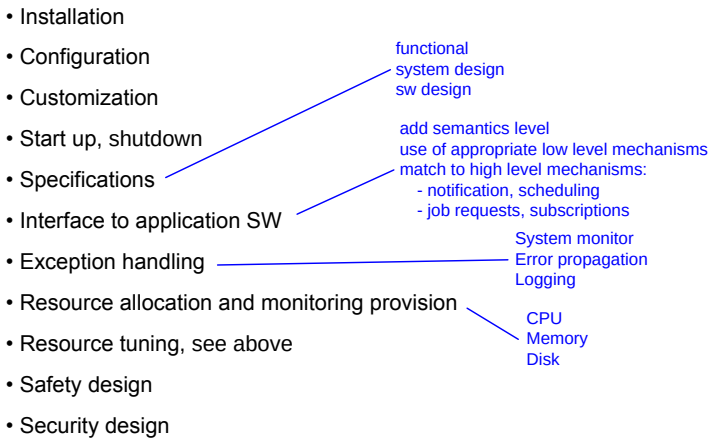


Figure 29: Embedding Costs of Purchased SW

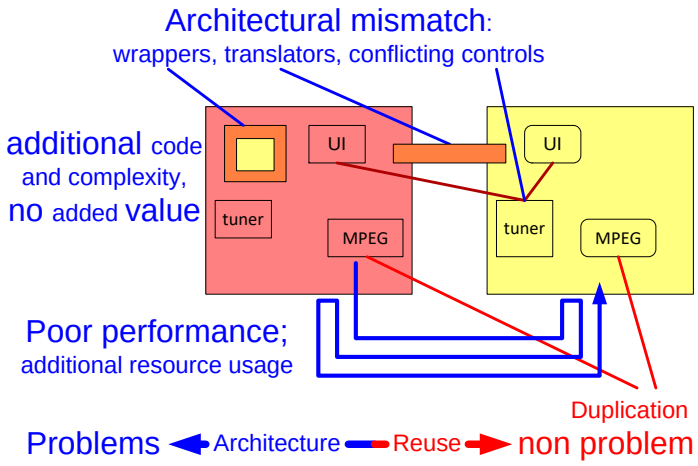


Figure 30: Example of Embedding Problems

5 The Time Dimension

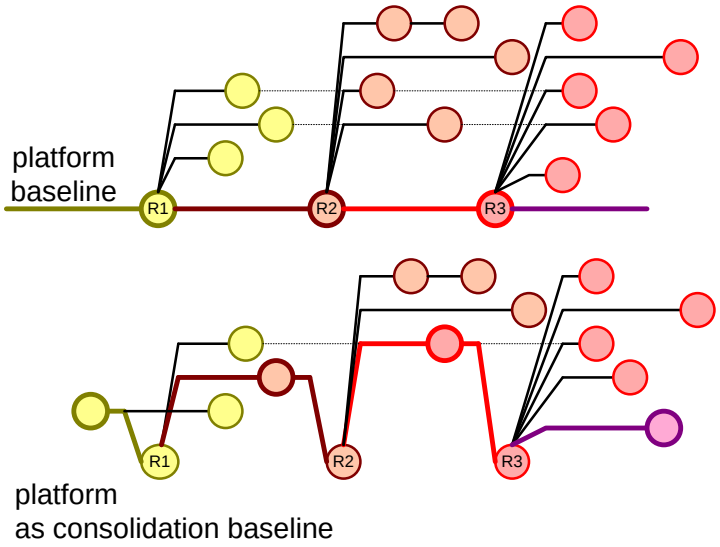


Figure 31: Who is First: Platform or Product?

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 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 32 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)

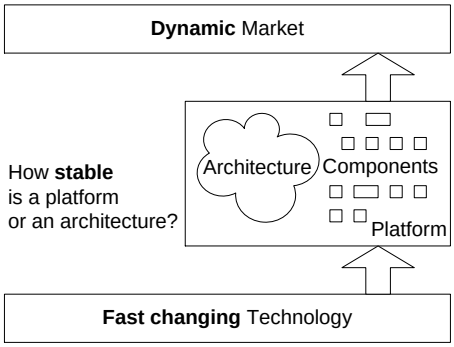


Figure 32: Myth: Platforms are Stable

- 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).

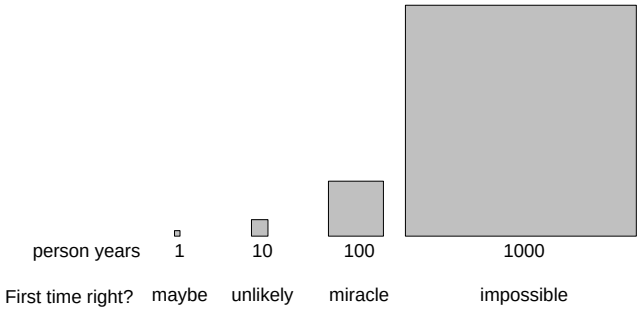


Figure 33: The first time right?

Figure 33 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 34 for a visualization of the influence of feedback frequency on project elapsed time.

The evolution of a platform is illustrated in figure 35 by showing the change in the Easyvision [7] 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

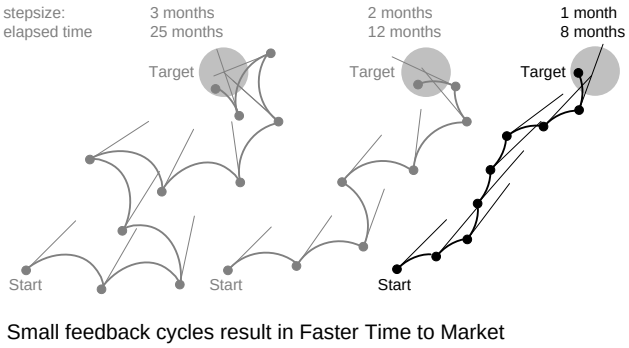


Figure 34: Feedback (3)

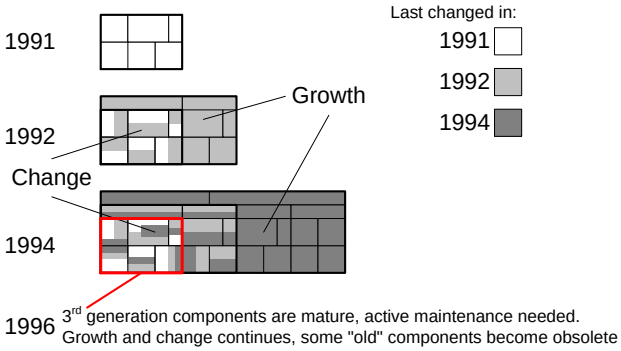


Figure 35: Platform Evolution (Easyvision 1991-1996)

code base is touched by changes.

The business context, the application, the product and it's components have all their own specific life-cycles. In Figure 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 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.

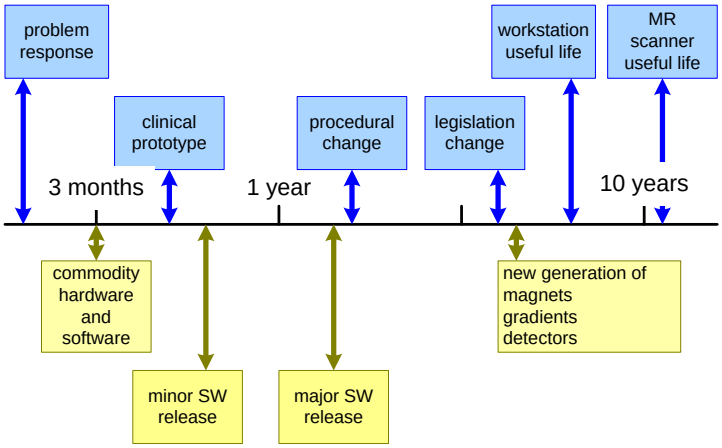


Figure 36: Life-cycle Differences

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

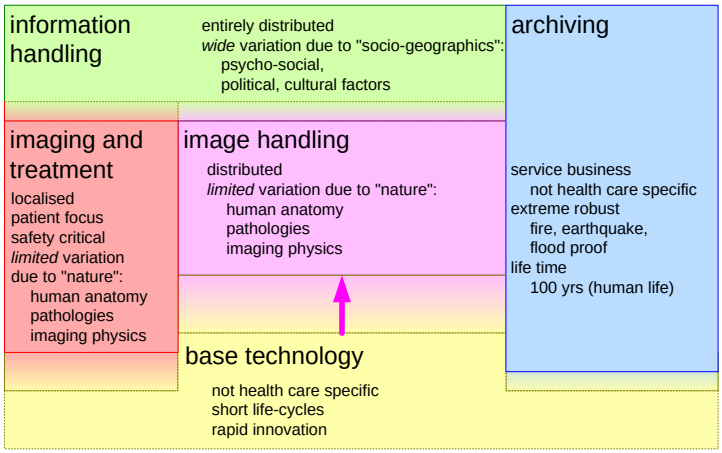


Figure 37: Reference model for health care automation

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[4]. 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.

[4]Today terrorist attacks need to be included in this list full of disasters, and secure needs to be added to the required qualities.

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 38.

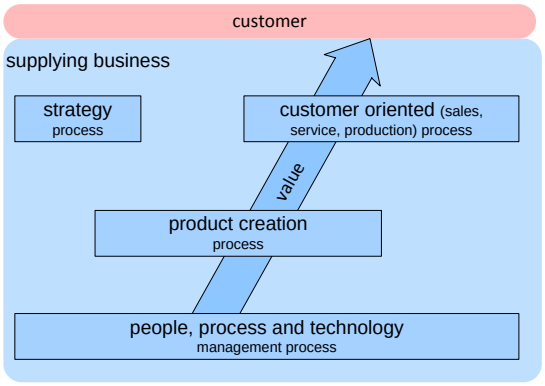


Figure 38: Simplified decomposition of the business in 4 main processes

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 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 39 shows the modified process decomposition

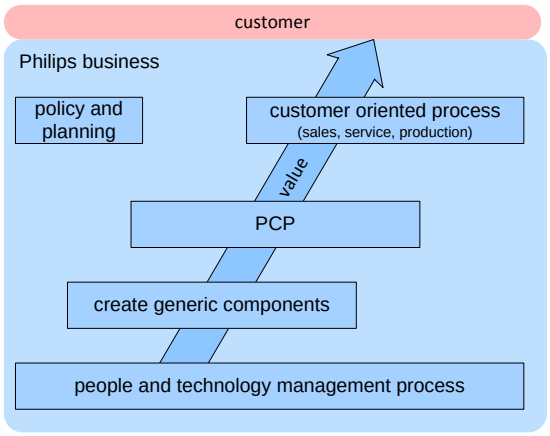


Figure 39: Modified Process Decomposition

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

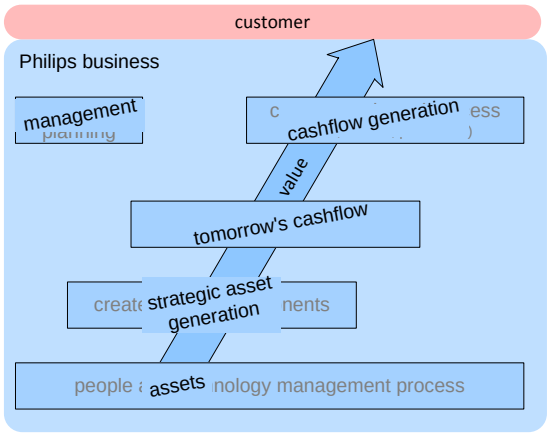


Figure 40: Financial Viewpoint on Process Decomposition

Figure 40 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 41. 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

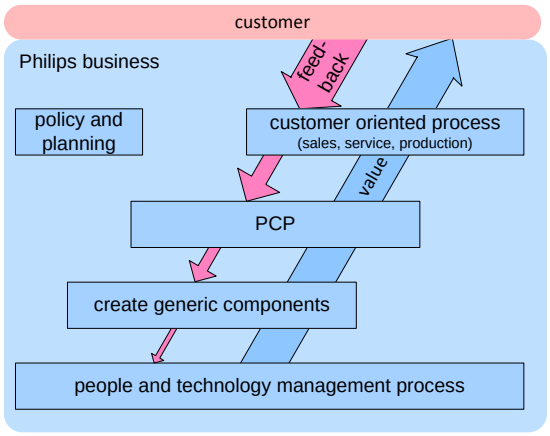


Figure 41: Feedback flow: loss of customer understanding!

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

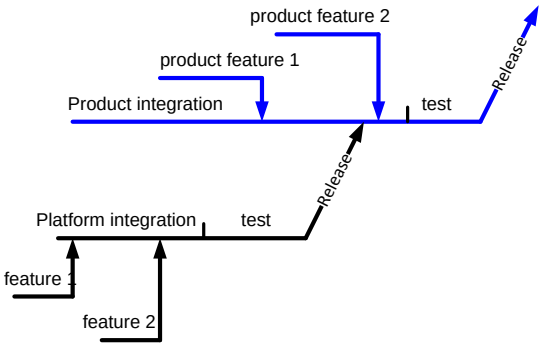


Figure 42: The introduction of a new feature as part of a platform causes an additional latency in the introduction to the market.

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 42.

The list of pitfalls in Figure 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

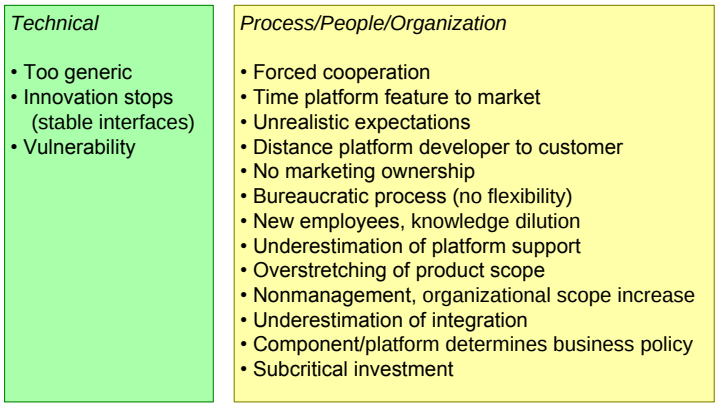


Figure 43: Sources of failure in platform developments

shown in figure 44.

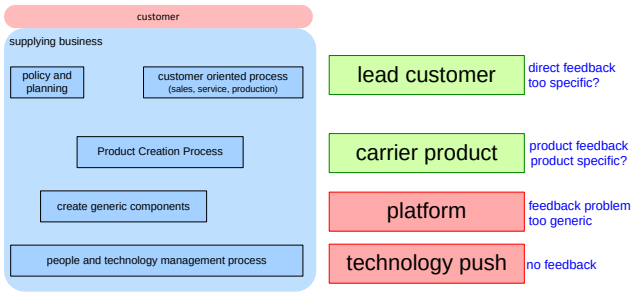


Figure 44: Models for SW reuse

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.

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.

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.

7 Market Driven

A useful top level decomposition of an architecture is provided by the so-called "CAFCR" model, as shown in figure 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!).

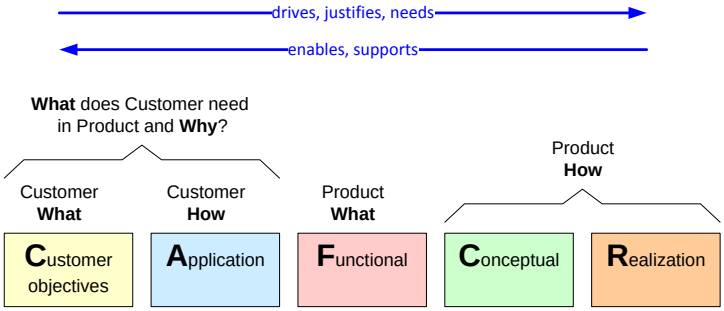


Figure 45: The "CAFCR" model

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 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

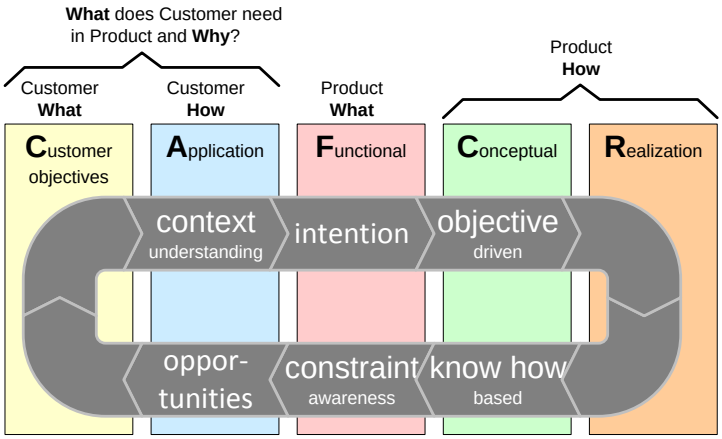


Figure 46: Five viewpoints for an architecture. The task of the architect is to integrate all these viewpoints, in order to get a *valuable*, *usable* and *feasible* product.

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 47 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

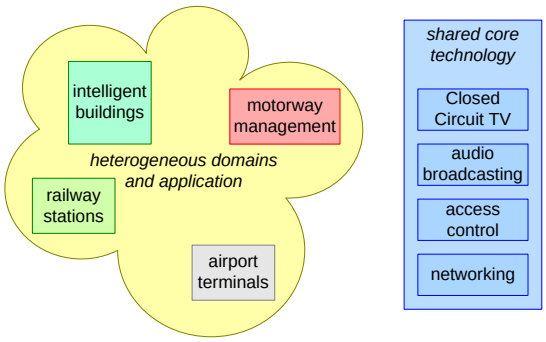


Figure 47: Example of Scoping of a Platform.

to find the right level of synergy is based on the key driver method [5]. 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.

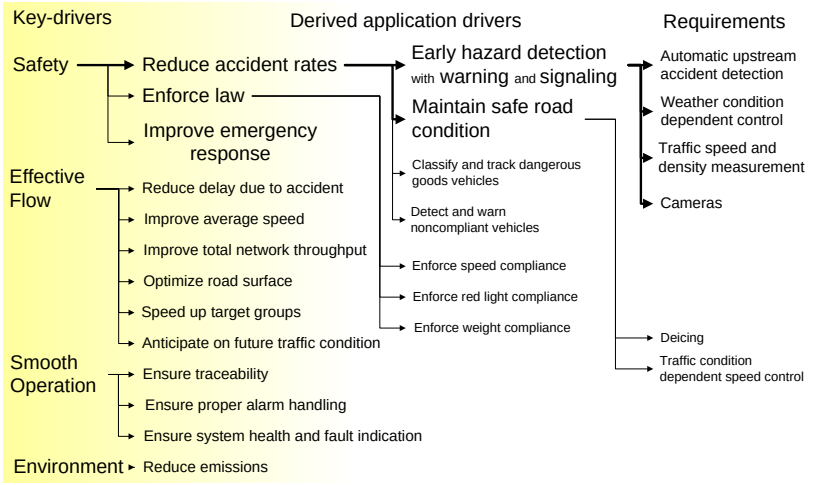
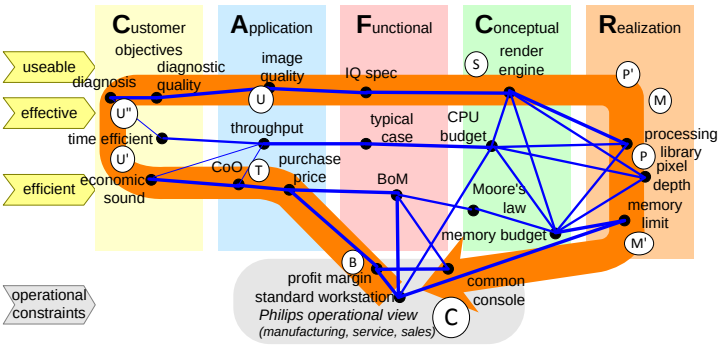


Figure 48: Example of the four key drivers in a motorway management system

Figure 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* [8] 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 49 shows an example of such a thread of reasoning for the Medical Imaging Workstation.



cost revisited in context of clinical needs and realization constraints; note: original threads are significantly simplified

Figure 49: Example Thread of Reasoning from the Medical Imaging Workstation

8 Recommendations

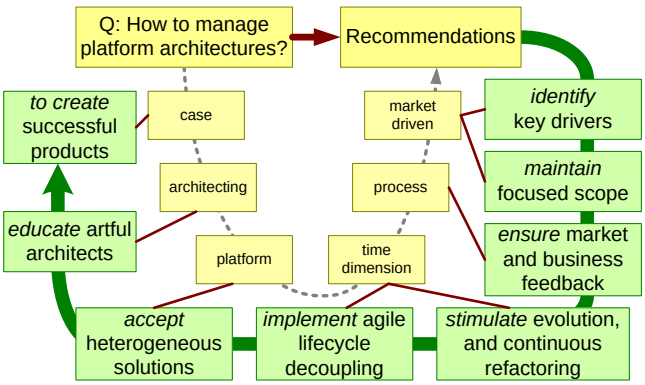


Figure 50: Summary of recommendations to manage platform architectures

Figure 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!

References

[1] Dana Bredemeyer. Definitions of software architecture. `http://www.bredemeyer.com/definiti.htm`, 2002. large collection of definitions of software architecture.

[2] Derek K. Hitchins. Putting systems to work. http://www.hitchins.co.uk/, 1992. Originally published by John Wiley and Sons, Chichester, UK, in 1992.

[3] 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.

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

[5] Gerrit Muller. Requirements capturing by the system architect. http://www.gaudisite.nl/RequirementsPaper.pdf, 1999.

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

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

[8] Gerrit Muller. CAFCR: A multi-view method for embedded systems architecting: Balancing genericity and specificity. http://www.gaudisite.nl/ThesisBook.pdf, 2004.

History

Version: 1.0, date: 14 June, 2005 changed by: Gerrit Muller

- Added text

Version: 0.1, date: 9 June, 2005 changed by: Gerrit Muller

- Added lots of slides and structure

Version: 0, date: 11 May 2005 changed by: Gerrit Muller

- Created, no changelog yet