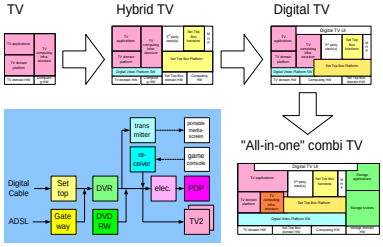
From Legacy to State-of-the-art; Architectural Refactoring

-



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

Abstract

The market of electronic appliances shows a fast increasing diversity. Manufacturers must be able to combine existing functions and new applications in a short time frame. A large amount of accumulated SW code (legacy) has to be reused in new ways.
The architecture(s) must be adapted to these new ways of working. Revolutionary adaptations have proven to be extremely risky. Opportunistic extension and integration decrease the quality of the code base, making it increasingly more difficult to continue. Architectural refactoring is a feedback based method to evolve an architecture.

version: 1.3 status: finished October 20, 2017

1 The problem

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 1 shows a typical set of today's audio and video products.

Figure 1: Today's Audio Video Consumer 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 2.

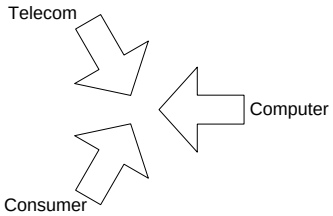


Figure 2: Trend: Convergence of separate worlds

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 3 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,

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

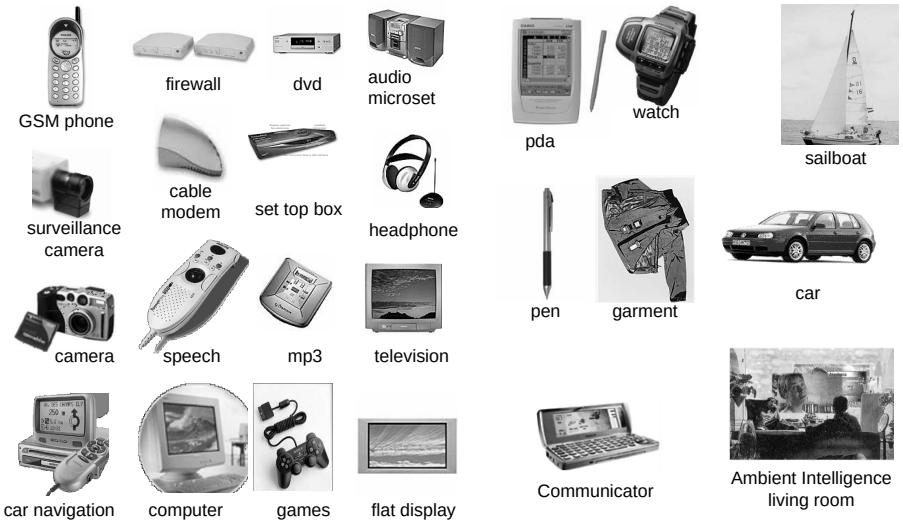


Figure 3: Integration and Diversity

In this presentation video entertainment will be used as the application area. Figure 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.

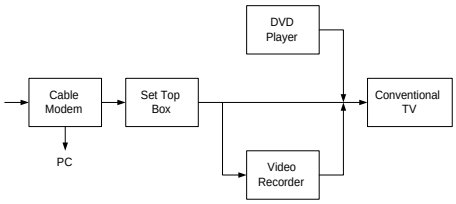


Figure 4: Today's Video Products

This chain of video products is slowly evolving, as depicted in figure 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.

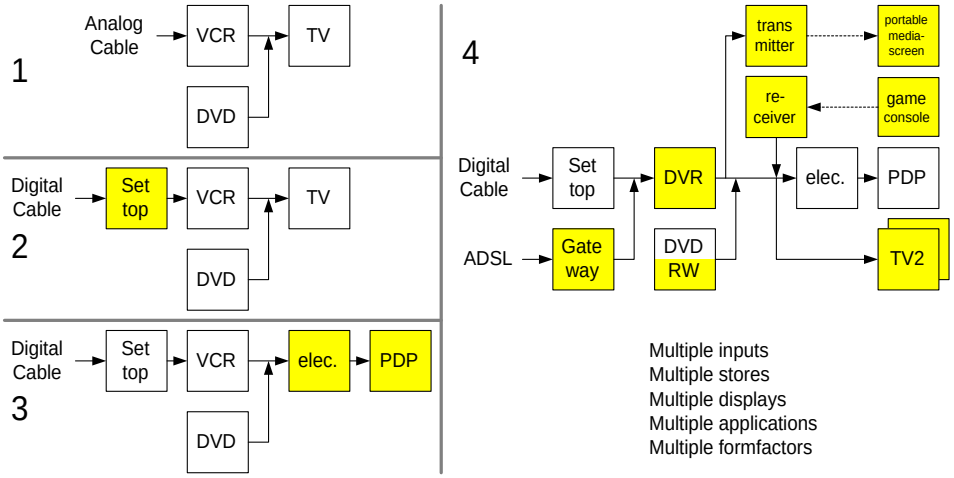


Figure 5: Evolution of Video Products

The function allocation for this last stage of figure 5 and the network topology can be solved in several ways. Figure 6 shows four alternatives, based on a client-server idiom.

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

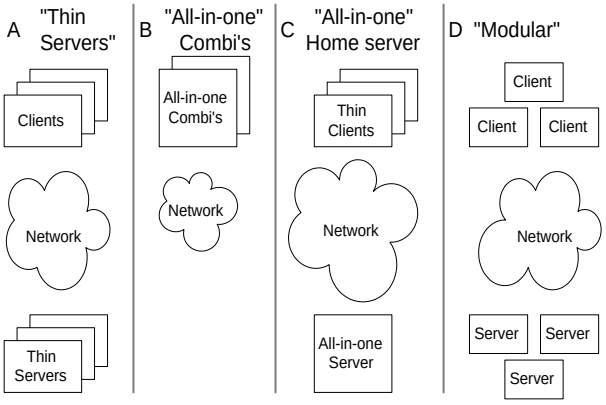


Figure 6: Distribution Scenario's

These alternatives require different packaging of functions into products, as shown in figure 7.

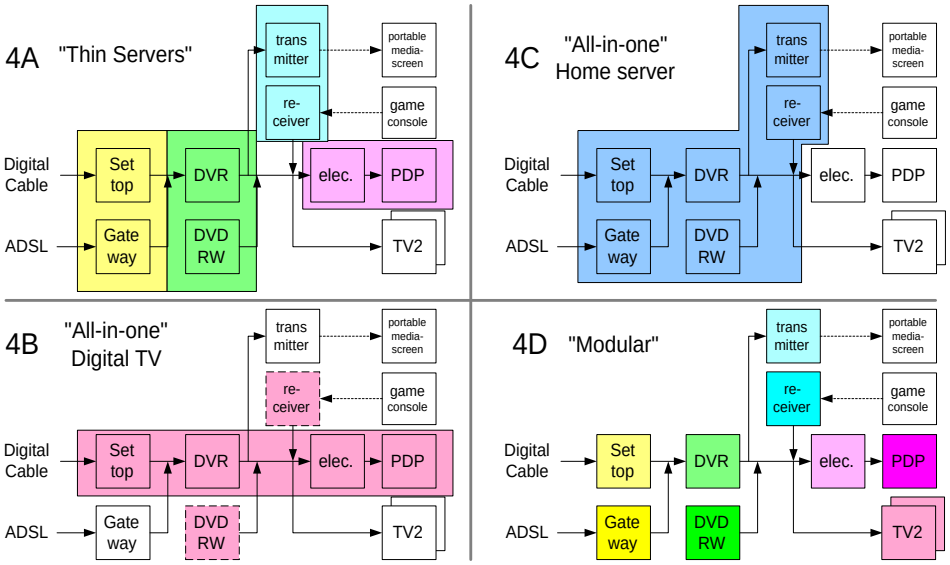


Figure 7: Product Packaging Options

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 8 shows the growth of the amount of memory in TV's.

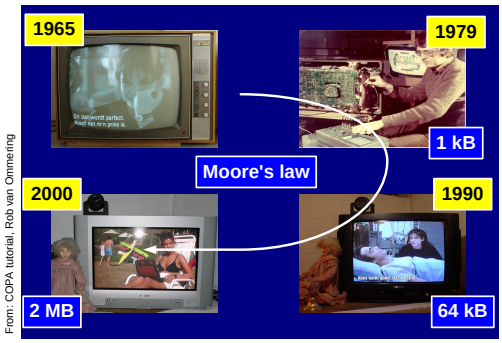


Figure 8: Moore's law

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

10 or more for ad hoc products. Typical values for CE type products is 3 faults per 1000 lines of code. See figure 9 for typical values as function of the year.

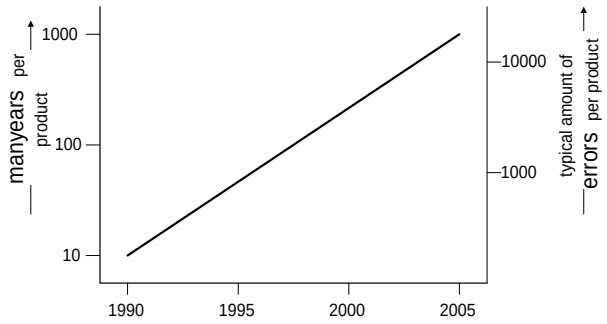


Figure 9: Problem: increasing SW size, decreasing reliability?

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

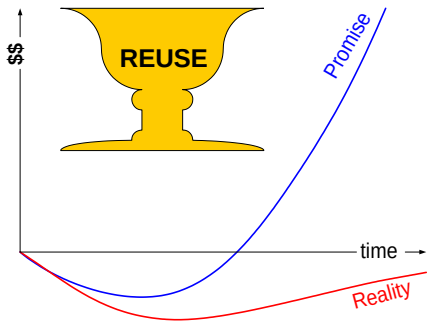


Figure 10: The Holy Grail: Reuse

Reuse is often presented as **the solution** for all problems mentioned above. Experience learns that quite the opposite happens in many cases, see [2], 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.

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 11 shows the simplistic architecting to achieve this merger.

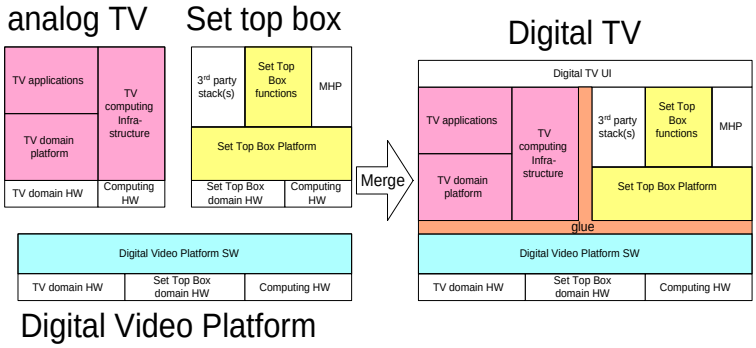


Figure 11: Simplistic Architecting: Digital TV

Figure 12 shows the rationale behind the reuse of existing software packages. The cumulative effort of the software involved exceeds 500 manyears.

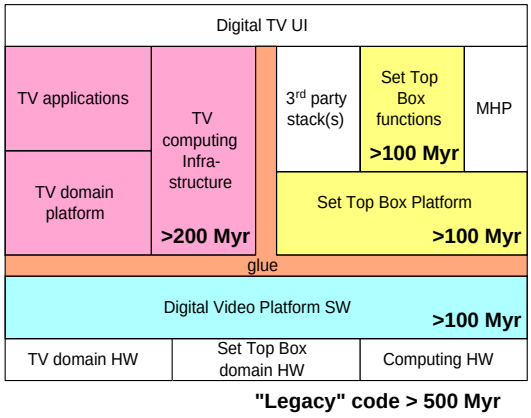


Figure 12: Available Code Assets

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.

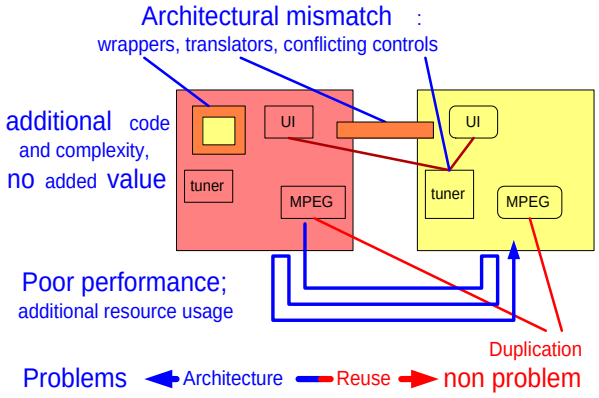


Figure 13: Merge problems

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 14.

Examples of Refactoring goals can be seen in figure 15. These refactoring goals should be sufficiently “SMART” to be used as feedback criterium.

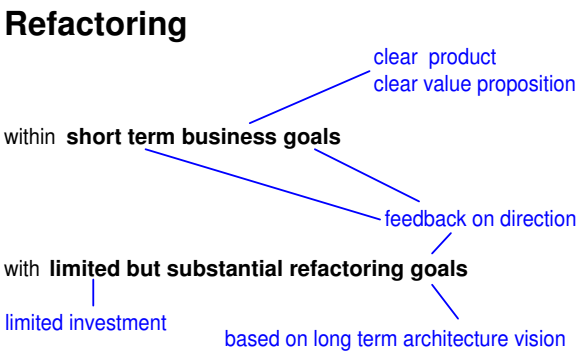


Figure 14: Solution: Architectural Refactoring

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 such a disaster.

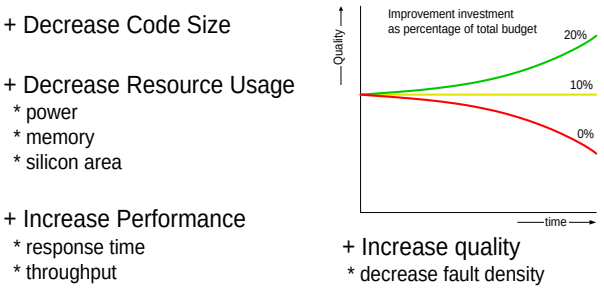


Figure 15: Example of Refactoring Goals

Architectural refactoring looks at all architectural aspects, from functions and structure to selection of mechanisms and technologies. Code refactoring, well known from extreme programming [1], plays a role at a much more microscopic level. See figure 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.

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

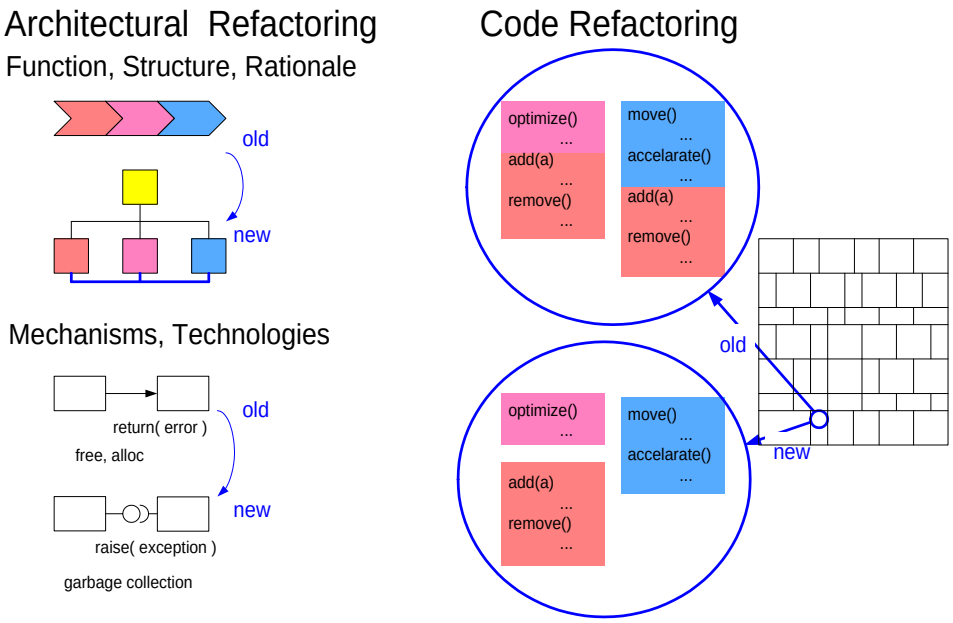


Figure 16: Architectural and Code refactoring

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 developments. 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 18.

The evolution of a platform is illustrated in figure 19 by showing the change in the Easyvision [4] 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 20 shows an example of a long term vision, where a framework is foreseen, which decouples 6 design and implementation concerns:

- applications

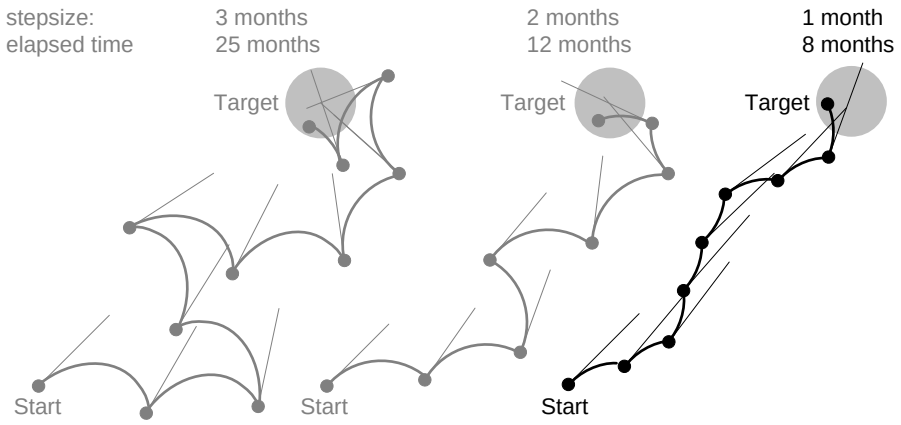


Figure 17: Frequent feedback results in faster results and a shorter path to the result

- services
- personalization
- 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.

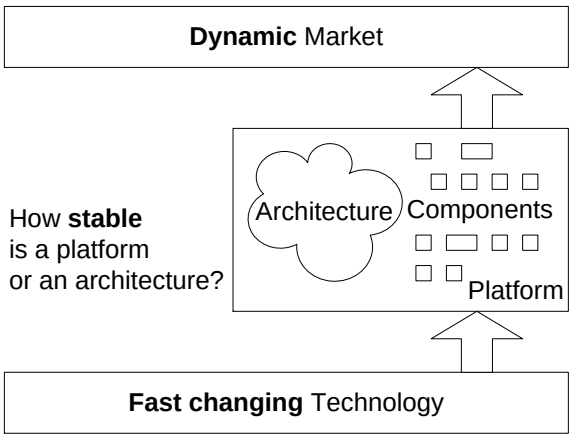


Figure 18: Myth: Platforms are Stable

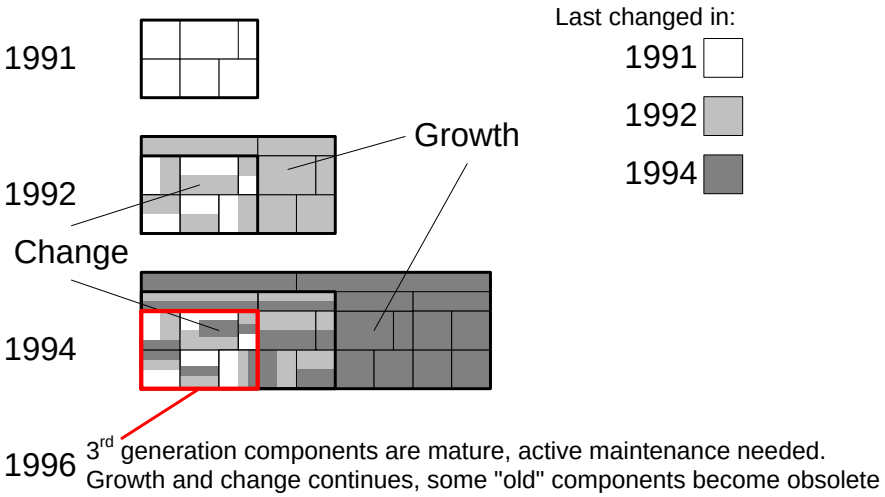


Figure 19: Platform Evolution (Easyvision 1991-1996)

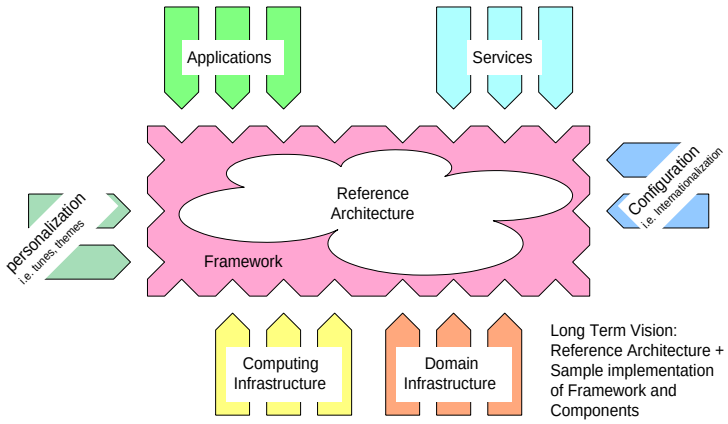


Figure 20: Example Long Term Vision

3 Conclusion

Figure 21 shows how **not** to work towards the future:

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

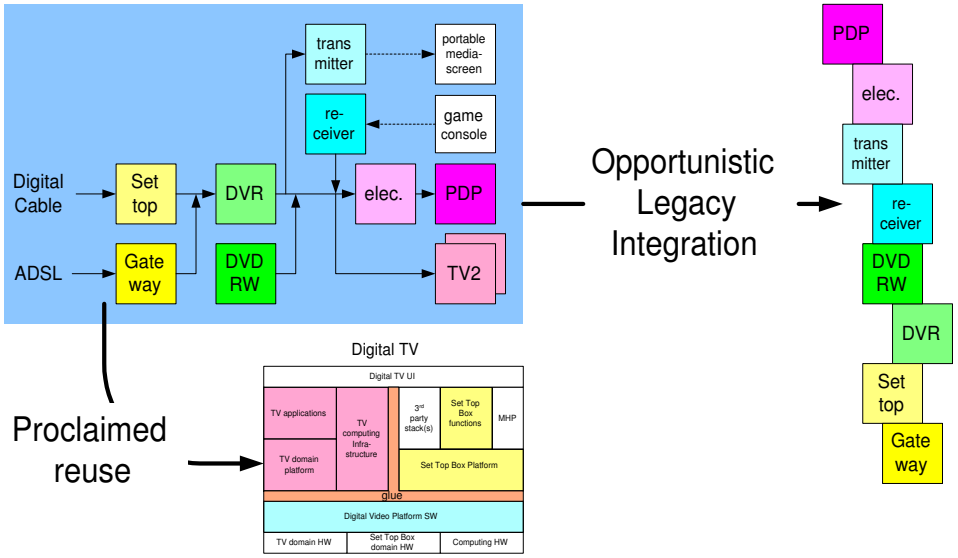


Figure 21: Don't do

While figure 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.

4 Acknowledgements

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

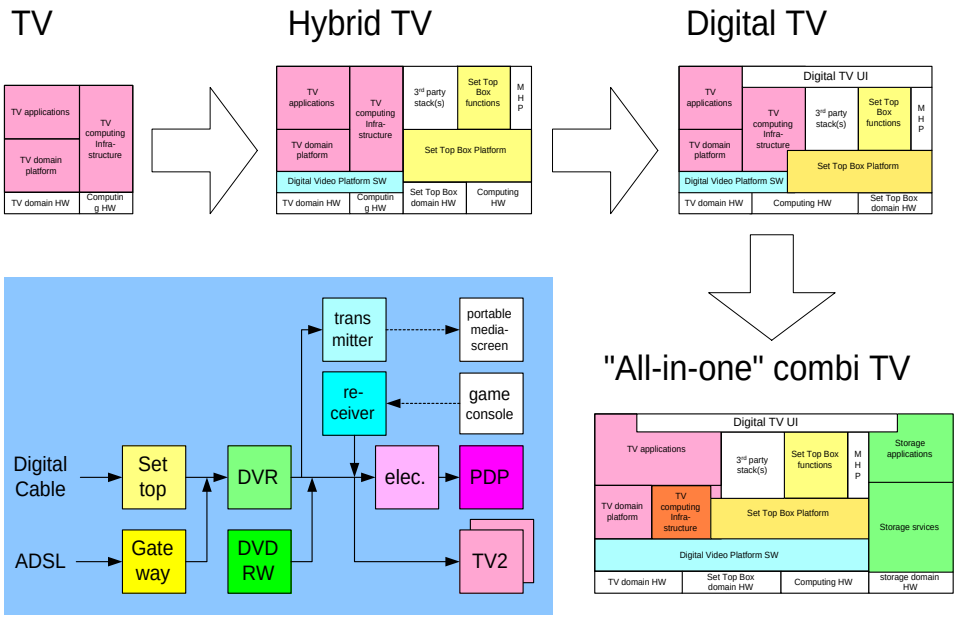


Figure 22: Conclusion: Refactoring the Architecture is a must

References

[1] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, MA, 2000.

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

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

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

History

Version: 1.3, date: June 13, 2002 changed by: Gerrit Muller

- minor change

Version: 1.2, date: September 12, 2001 changed by: Gerrit Muller

- "long term vision" sheet added to presentation

Version: 1.1, date: September 6, 2001 changed by: Gerrit Muller

- Created, no changelog yet