14

Philips Consumer Electronics Software for Televisions

with Rob van Ommering

Company facts of Philips TV Software

Organisational size: 250 developers.
Starting Mode: New architecture, reverse engineered code.

Experienced improvements:
- A single software product line for all of Philips' mid-range and high-end television products.
- Able to produce the variability desired by marketing.
- Variability no longer a key problem for architect.
- Software development no longer on the critical path of product development.
- Still no need for a new software architecture after six years, while previous architectures lasted less than five years.

Business: To support the required variability, while maintaining a high quality-level and enabling combi-products in the future.

Architecture: A compositional rather than a decompositional approach is taken. The Koala component model and architecture description language is tuned towards use in resource-constrained systems.

Process: A change from a project organisation to a products and assets organisation.

Organisation: A product-oriented organisation was changed into a single development organisation that hosts asset and product teams.

14.1 Introduction

Philips Consumer Electronics, a division of Royal Philips [103], is one of the largest consumer electronics companies in the world. It has an annual turnover of €10 billion and a sustainable profit of 5%, which is considered quite well in this domain. Philips Consumer Electronics has 16,000 employees and is present in all regions of the world.

Televisions are responsible for one-third of the turnover of Philips Consumer Electronics.[1] Though a quite traditional product, they are an important factor in shaping the brand image that will allow all Philips divisions to create and enter new markets in lifestyle, healthcare and technology. Philips has a market share of 10% in televisions, roughly equal in size to its main competitors.

In this chap. we study the software product line that was set-up to create the software for televisions. The technology for this product line was created in 1996, the product line itself was initiated in 1998, and the product line has been in actual use since 2000. The approach is extensively documented in [148].

14.2 Motivation

Televisions were one of the first consumer products to contain embedded software and hardware. This started with an 8-bit micro controller and 1 KB of memory in 1978, and since then both hardware and software have grown following Moore's Law quite closely [29]. Fig. 14.1 shows the size of software in high-end televisions as a function of time.

The most important reason for this growth is the continuing integration and miniaturisation of hardware, with an accompanying decrease in costs. This allows to implement more and more of the functionality in software:

- A large part of the control shifted from hardware to software, for instance setting the tuner, detecting stereo sound and blanking the screen during zapping.
- Software made it possible to create fancy user interfaces, starting with character-based On Screen Displays, followed by character-based menus, then 2D graphical menus and now moving to 3D graphical menus.
- Data processing in a TV (Teletext) was done in hardware at first, but is now mostly done in software; only the basic capturing of data is still done in hardware. Other examples of data processing are closed captioning[2] and electronic program guides.

[1] From here on, the term "Philips" is used as a short hand for "Philips Consumer Electronics"

[2] Closed captioning means displaying a transcript of the audio part of a television program

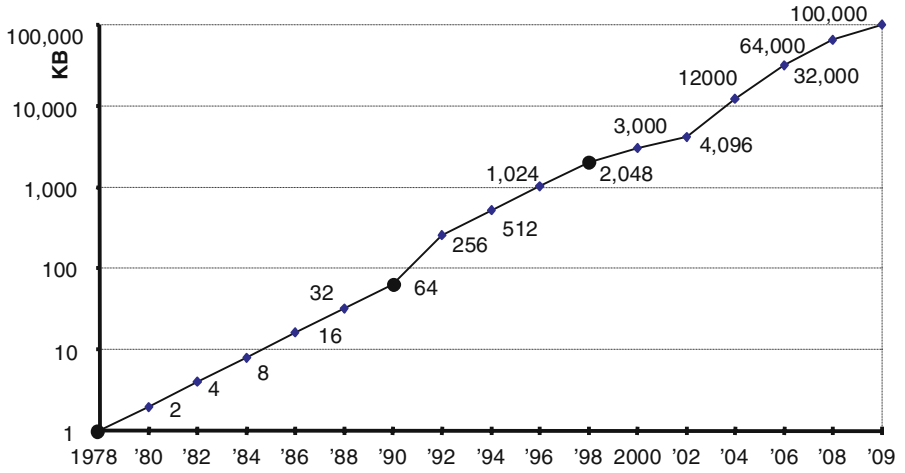

Fig. 14.1. Growth of software (code+data) in high-end televisions (in KB)

- Sound processing – decoding, featuring and rendering – has shifted to software quite a few years ago already, and image processing has recently shifted to be implemented mostly in software.
- Modern digital standards such as MPEG-4 make processing in software obligatory.

While this trend may alleviate the design of hardware to some extent, it certainly makes the design of software more complex. Managing this complexity is one of the challenges that we are still facing: a television is roughly as complex as a personal computer ten years ago.

The second challenge stems from the need for the company to bring out its products globally. While the global market was very diversified at first, around 1996 the common parts of the functionality in a TV grew larger than the region-specific parts, making a product line approach feasible. But profiting from this phenomenon and reflecting this commonality and variability in software is a non-trivial task, as many companies have experienced.

The third challenge was the upcoming convergence of products. Prototypical example of a convergence product around 1996 was a TV with built-in VCR, which allowed features such as a one-button "record what you see". The first such product consisted of two separate hardware and software systems that were very loosely integrated. To become more cost-effective, the two software architectures had to be integrated to run on a single CPU and in a single memory. More convergence products were expected, such as a TV with built-in DVD. A significant part of the problem was that TV and VCR (later DVD) software was developed in different divisions, each with their own profit and loss responsibility.

These three challenges combine to the following problem statement. The complexity of software is growing, and the number of product types increases, while the lead-time must decrease and the quality of the software must be maintained (consumers do not expect products to crash regularly). Philips took a number of actions to achieve this:

- *Urgent*: achieve "reuse in space", leading to an almost classical product line approach.
- *Medium-term*: achieve "reuse in time": making sure that products with new features can be produced every year.
- *Long-term*: solve the convergence problem.

Reuse in space involves properly managing the diversity of complex software in a product line. Obviously this involves more than maintaining a simple list of variability parameters: there will be hundreds of such parameters so at least some form of hierarchy is needed, and also the structure of the software will depend on variability.

Reuse in time requires evolution rules that dictate how parts of the software may be changed without breaking other parts of the software. This may seem a simple "backward compatibility issue", but there are many subtleties involved that make this very difficult in practice, as anyone will understand who has upgraded his operating system to a newer version and found his favourite applications not working anymore. This issue includes a proper anticipation of changes in hardware and coping with this in software.

But even if we manage to reuse 100% of our software over time, that will not solve all problems. Because in a world following Moore's Law, that would only delay our problems by two years.[3] The more fundamental solution is to obtain software from elsewhere. Not by outsourcing it – as that solves the people problem but not the cost – but by getting it from vendors who leverage their development costs over multiple customers.[4] For a consumer electronics company, part of the software can be obtained from the hardware supplier (the semiconductors company), and part can be obtained from independent software vendors.

The convergence problem is the hardest to solve, as it also involves crossing organisational boundaries within a company. The technical solution is to use composition instead of decomposition. To address the organisational issues, the existing situation of loosely coupled product teams was changed into a single development organisation, with asset- and product-oriented teams.

[3] Imagine a new product coming out every two years, where the size of the software has doubled. Even with perfect reuse, half of the software will be new. This implies that one still needs a team that grows exponentially in time, only two years shifted in time

[4] Thus establishing reuse over company borders

14.3 Approach

Before we delve into the technical details of the product line approach, we will first describe how the product line was actually set up.

In 1996, software architects in the Philips TV department foresaw severe problems in managing variability and asked Philips Research for a solution. By that time, there was already a long-standing co-operation between the TV department and Research in managing the ever-growing complexity of software, which had resulted in the software architecture that was used then.

Philips Research responded by comparing different software component models and creating Koala from the most suitable elements of these to solve complexity and variability issues in resource-constrained systems. This component model was transferred then, but using it to create the next generation software architecture turned out to be difficult while also maintaining the current architecture: key people could not be freed to work on the new architecture without endangering the current set of products.

Therefore, Philips Research was asked to set-up the next generation architecture and to fill this by reengineering the existing code. Interestingly, the resulting software architecture outlived the original hardware architecture for many years!

Research spent one year in setting up the architecture (1998), and then one year to build a lead product with this architecture together with the TV department (1999). The choice of the lead product proved to be critical: a product was chosen with high visibility and low risk. The lead product actually failed for non-technical reasons, but the second lead product was successful (in 2000), and within two years the software architecture and accompanying approach was used in the full range of Philips" TV products.

In the first two years (1998–1999), most of the developers of the TV department were still maintaining the old architecture to bring out the majority of products (and thus generate income). After that, developers gradually moved from the old loosely coupled product teams into a new single development organization.

Choice of the team also proved to be a critical success factor. The initial architecture team consisted of three architects, one experienced in software, one in business aspects and one in the domain. Five high-quality developers were soon added, experienced either in the domain or in software engineering (or both). An experienced project leader was added too.

As important as the team itself were the champions in the product division, monitoring and defending the new approach. The direct owner of the research work was the development manager of the TV department. The research was sponsored by the software director and monitored by the software process manager of Philips.

14.4 Business Aspects

The television market is an established market: it does not grow much, but it is very important for a company such as Philips to maintain market share, as this provides 10% of the Philips turn-over (30% of Philips Consumer Electronics). Also, it provides an important brand image for Philips to sell a variety of other products.

This market shares some typical characteristics with other consumer markets. Features initially introduced in high-end TVs soon become commodities, i.e. they are not positively discriminating anymore but they are *must-haves*: they negatively discriminate a product that lacks them. To maintain market share, development should be focused on adding new features, rather than on reimplementing old features.

The television market is a global market with quite some regional variability and a large range of prices. While individual products last over ten years, waves of new technologies tempt customers to buy new products sooner: black and white to colour, Teletext, sound and image quality, 100 Hz, 16:9, flat displays, picture browsing and connectivity are examples. The pace of introduction of new products onto the market increases from yearly to half-yearly or even shorter. Competition increases since PC and display technology make it possible for other companies, such as Hewlett-Packard and Dell, to enter the TV arena.

The short-term challenge for software development is to stay away from the critical path, to support as much variability as marketing requires and to maintain the quality level required for consumer products. The long-term challenge is to enable convergence in the form of combi-products and to compete with the PC industry that is trying to capture the living room.

14.5 Architecture

Many researchers in the field of software product lines believe in an a priori analysis of commonalities and differences of products in the portfolio, followed by the creation of a single reference architecture with explicit variation points.[5] While this may work for a small product line of TVs, we were in doubt whether it would work for the whole range of TVs that Philips produces, and we were sure that it would not allow us to create combi-products, if only because it is very difficult to agree on a common software architecture between different product departments.

We therefore opted for a *compositional* rather than a *decomposition* approach, partly inspired by the way that this was already possible in hardware. We designed the Koala component model, inspired by existing models such as Microsoft COM and Darwin [89], but tuned towards use in resource-constrained systems.

[5] See [11] for a discussion

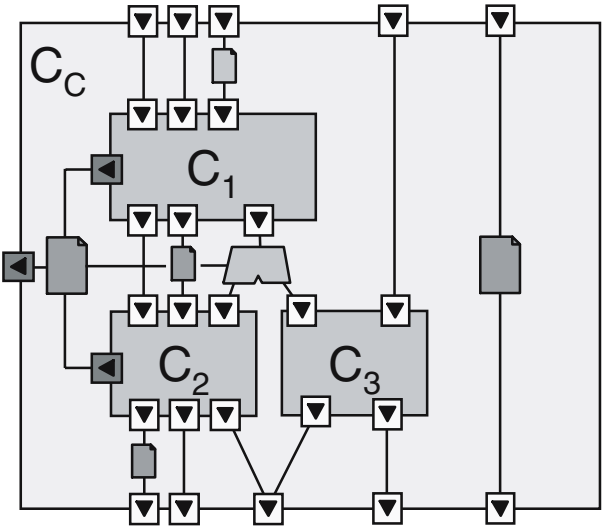

Fig. 14.2. An example Koala software component

Figure 14.2 shows an example component in Koala that illustrates many of the features of the model. First of all, a component is a unit of design not only of reuse, but also of implementation. In plain words, this means that a component has a description of the interfaces at its borders so that it can be used in various contexts, but it also has a specific implementation that cannot be separated from the component: a component is a specification and an implementation.

Koala components are implemented in C. The document-shaped objects in Fig. 14.2 represent C files. The squares with embedded triangles represent functional interfaces, with the triangle pointing into the direction of function call. A component not only specifies the interfaces that it *provides* to its environment, but also the interfaces that it *requires* from its environment: all communication with the environment is routed through interfaces. Configurations of components where the required interface of one component is provided by another one again form components. Thus, the component model is hierarchical.

In the file system, a component is a directory with a set of (C) source files and a file containing a Koala component description. There is no makefile for the component: the makefile is automatically generated from the component description. Put differently, the component description takes the place of the makefile in traditional development.

Interfaces are defined, syntactically and semantically, in separate files and in a separate language – the Interface Description Language (IDL) part of Koala. This allows us to reuse interface definitions to create multiple implementations of a functionality (remember that each implementation is a

separate component). Also, by using an interface model similar to that of Microsoft COM, we can extend components with new functionality without breaking existing applications of that component. The Koala interface mechanism does not incur extra cost at run-time, as most interface bindings are resolved at compile-time. Interface binding only results in run-time code if the binding cannot be determined at compile-time.

An important side effect of the use of an IDL to specify interfaces is that interfaces are kept relatively clean. This includes a proper separation of type definitions from functional interfaces and a proper separation of a functional interface from an inline implementation. In classical C, these two facets are combined in a single header file.

Because configurations of components are again components, a product is a decomposition tree of components. Note that a product line is a composition *graph* of components, as basic and compound components can be used in multiple products. Maintaining this graph is the main task of the architect.

The composition graph is actually the high-level mechanism to deal with variability: different products may have different sub-systems sharing (a subset of) the same components. Diversity parameters[6] provide a low-level variability mechanism to parameterise code. These diversity parameters are organised into the so-called *diversity interfaces,* which are treated as ordinary required interfaces. Switches allow to implement structural diversity in the configuration of sub-components as an internal variability of the compound component (steered by diversity parameters), thus forming the bridge between the "high-level" and the "low-level" variability mechanism.

Packages group components and interfaces into private entities for use within a team only, and public entities for use by other teams. Packages are hierarchical: sub-packages allow a team to internally structure complex packages, while super-packages allow to model dependencies between packages in a hierarchical way.

The television software is divided into three layers – all made by Philips initially – with standard APIs between them (Fig. 14.3). This architecture had a clear intention: on the longer term, the operating system (OS) part

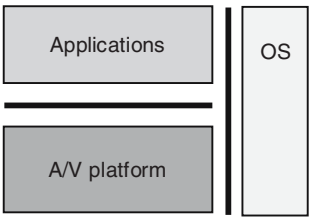

Fig. 14.3. Basic division of television software in three layers

[6] 'Diversity parameter' is the Koala terminology for a 'configuration' - or 'variability' - parameter

would be obtained from independent software vendors, while the audio/video (A/V) platform would be obtained from the hardware supplier, leaving only the applications to be developed by the television department.

The use of an architectural description language (Koala) enables enforcement of certain rules, such as allowed usage relations between packages and easy checking of other rules, such as multi-threaded use of interfaces. The ADL has a textual and a graphical syntax: the textual syntax is used in the product generation, while the graphical syntax is used in design discussions. Approximately 10% of the code is written in Koala; the rest is in C. The Koala compiler generates C code and header files from the Koala descriptions. The graphical component diagrams are made either by hand with Visio, or by using a tool that automatically generates a diagram from textual descriptions.

14.6 Process

We have described some of the technical aspects of the software product line approach in the previous sect., and they are indeed important success factors. But a development department cannot switch to a product line approach by incorporating technology only. The development process and organisation has to be changed as well. We found the most important change to be the move from a project organisation to a products and assets organisation. Previously, there was one large team per product, doing a complete waterfall life-cycle for that product. In the new way of working, medium-sized asset teams have an iterative development process, and small product teams integrate assets.

Of course, work is still organised in projects, but there are now two kinds of projects: long-running projects to develop and evolve assets and relatively short-running projects to build products. Figure 14.4 shows the relation between these types. There are various models for organising asset and product development, but we found the most workable to be an *m:n* relation between sub-system and product development: sub-systems working for m products, and products integrating n sub-systems. This relationshop between products and sub-systems is defined by the architecture. Other models have a platform integration step in between, but this adds an extra delay between implementing a new feature and utilising it in a product.

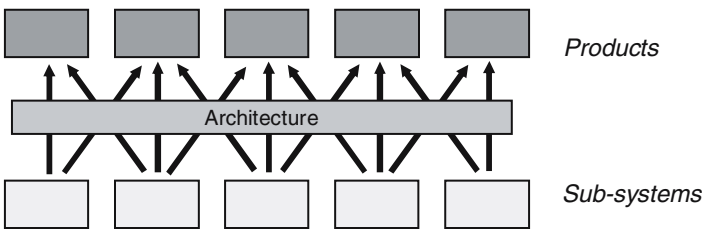

Fig. 14.4. The relation between sub-system and product development

The way software is documented also has to change. A traditional, project-oriented software development organisation typically creates requirements, global design and detailed design document, describing what is going to be made. Our product line documentation includes component and interface data sheets, describing what has been made, with additional architectural sub-system design notes, forming a living description of the evolving architecture.

Configuration management is traditionally implemented as one big archive for all software for all products, creating complexity and performance problems, or as separate archives for small sets of products, with opportunistic reuse only between the different archives. We found it convenient to have a configuration management system per asset team and per product team, and have weak links between these archives, exchanging formal releases of the software only. This scales relatively easily to larger organisations.

We are also in favour of an open development environment[7] where everyone can see every bit of software. We achieve this by publishing all releases on the intranet. Of course, there are parts of the software for which this is not possible, e.g. due to licensing reasons.

Testing also has to be adapted. Traditionally, products are created incrementally, with continuous system testing ensuring that new features work and do not break existing features. But if sub-systems are to be used in multiple products, they have to be tested in isolation and this also holds for components within sub-systems. However, testing sub-systems in isolation is a non-trivial issue: many problems become apparent only after integration with other sub-systems. This is why we create pseudo products at an early stage: products that are not to be released on the market, but that serve as early integration test for (new versions of) the sub-systems.

Managing requirements for a product line is a challenge by itself: instead of separate requirement documents per product, one would like to optimally profit from commonality between products, but at the same time keep individual documents readable. We also found that improperly structured requirements documents, where the product line aspects were not taken into account usually result in improper design structures. While this can be solved at the design level, the harm is still done at the requirements level.

As a final remark on the consequences that a product line approach has on the development process, the fact that sub-systems are used in multiple products implies a road-mapping activity that plans and agrees on deliveries between asset teams and product teams. In our organisation, a release matrix is used to maintain a view on releases as functions of time.[8]

[7] That is, open within the company. This is sometimes called Inner Source

[8] See [138] for another discussion on this topic

14.7 Organisation

Business needs must be translated into an architecture, and the architecture shapes the development process. To execute this process effectively, the development organisation must be adapted as well.

In Philips, software development was traditionally organised in product teams responsible for the development of a particular product or a small set of such products. We changed this in a number of steps into a single development organisation that hosts asset teams and product teams.

The asset teams are in principle funded from a single source, derived from the sum of contributions of the product teams. This is possible because there is a single development manager heading all developments. Although in theory asset teams work for multiple products, we have seen cases where product teams requested specific sources from asset teams to work on their product. While there is nothing wrong with asset teams being very aware of the specific products that use their sub-system, we generally object to "people shopping".

The balance between generic and specific development is delicate. In principle, asset developers have the long-term applicability of their assets as goal, while product developers have the successful release of their product onto the market as immediate goal. But long-term assets have no value if short-term products fail to be on the market in time. We have seen various models to successfully build products in our organisation, including asset developers joining product teams for short periods of time, and asset archives being split into product specific branches. This may be appropriate in certain cases, but there must also be a force pushing towards product independence and long-term value of assets. At the end of the day, the only way that we managed to achieve this is to make it the personal responsibility of the asset teams, and reserving sufficient resources for asset development.

Another complicating factor is distributed development. TV development sites were traditionally distributed around the world, with many locations in the USA and in Europe. Development sites have also been opened in India and in the Far East. We found it important to align the architecture, the project structure and the organisation. In plain words, a sub-system is developed by a single (asset) team located at a single site. We initially had four cases where this alignment was not optimal – i.e. distributed teams were working on a single sub-system – and in all cases the result was severe communication and integration problems. Others report similar experiences [153].

14.8 Results

While in 1996 variability ranked high on the list of issues of the software architects, variability has completely disappeared from that list since the introduction of Koala and the accompanying product line architecture and approach. The team is now able to create the diversity of products required by marketing, and to produce these different variants on time.

An interesting side effect is that architects who joined the team in 2002 or later have not experienced variability problems themselves and sometimes fail to see the added value of a solution like Koala. Since it always costs a certain amount of effort to maintain a proprietary solution, some want to remove it altogether. This shows how a success factor can become a failure factor later. Replacing a proprietary solution with a solution obtained elsewhere is a different issue, and should be done as soon as such a solution is available, and the cost of conversion is not too high.

Although the initial lead product failed for non-technical reasons, the second product succeeded. After that, there was a quick ramp-up and, within two years, all of Philips" mid-range and high-end televisions were produced with this approach.

Another indication of success is the fact that so far there has been no need to develop a new architecture. Previous architectures lasted at most five years. Of course, with software size still growing according to Moore's Law, setting up a new architecture from scratch is almost unaffordable.

14.9 Lessons Learned

The most important lesson that we learned is that all facets of BAPO must be addressed otherwise the introduction of a software product line approach will fail.[9] There must be a business drive to create a product line, there must be a proper software architecture and component technology, the development process must be adapted and the organisation must be made to fit.

It took us three years to become successful, and even longer if you measure elapsed time. We sincerely doubt whether the introduction of a product line approach in this context could have been done any faster, given the amount of change that was needed in architecture, technology, process and organisation. But keeping such an activity alive for three years without intermediate results is very difficult, and has failed in some other parts of Philips. A technique that can be introduced incrementally would therefore be highly welcomed. But we also have experiences that changing an architecture incrementally is a process that may be too slow to obtain significant results within a few years.

A successful product line organisation requires a very delicate balance between application and domain engineering. The introduction of a similar approach failed in another part of Philips. After setting up the structure, the organisation could potentially produce many products, but fell in a genericity trap and did not succeed in creating even a single product. As a result, the organisation changed back to a structure that successfully produced a single product but the potential for creating a product line was mostly removed.

One of the failure factors of a product line organisation is to have too many dependencies among deliverables. If a change in sub-system A has to

[9] In fact, the case described here was one of the inputs in the creation of the BAPO approach

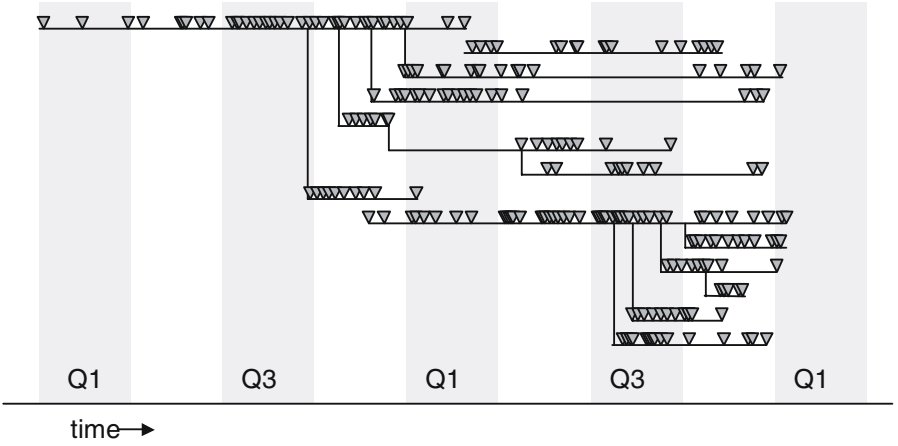

Fig. 14.5. Actual branching of sub-systems

be integrated into sub-system B before B can be released to C, then the time between fixing a problem or implementing a new feature can become too long. The *m:n* delivery model as described above is better, but requires strict evolution rules and still some form of pre-integration.

The configuration management strategy is another critical factor. Many configuration management systems claim that they can manage variability, but they can do this only at the level of files, and they can only handle compile-time variability. It is better to solve variability in the architecture, and use a traditional configuration management system for version management and for temporary branches to safeguard a product that is to be released from changes to sub-systems made on behalf of other products. This scheme worked fine for us initially, but then temporary branches were kept alive increasingly longer, resulting in many parallel branches to be maintained. Figure 14.5 shows an example of this. Each vertical band represents a quarter of a year, the horizontal lines represent different branches of development, and the triangles represent (intermediate) releases. The longest living line is the main line of

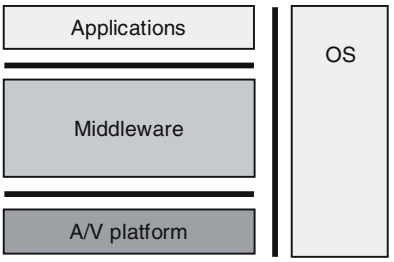

Fig. 14.6. The top-level architecture as it evolves

development, the rest are side branches. We measured the amount of effort spent on side branches and found this to be less than 15% [149]. Therefore, we still regard our product line approach as being successful, even though improvements are still possible.

Figure 14.3 showed our initial top-level architecture, based upon the idea that the three parts would be implemented by three different parties in due time. This is in fact currently happening, with one addition: between A/V platform and applications a middleware layer is emerging, with software provided by an ecosystem of independent software vendors, which again reduces costs by leveraging the software over more products, as shown in Fig. 14.6.