university of groningen

University Medical Center Groningen

University of Groningen

Building Product Populations with Software Components

Ommering, Robbert Christiaan van

IMPORTANT NOTE: You are advised to consult the publisher's version (publisher's PDF) if you wish to cite from it. Please check the document version below.

Document Version
Publisher's PDF, also known as Version of record

Publication date:
2004

Link to publication in University of Groningen/UMCG research database

Citation for published version (APA):
Ommering, R. C. V. (2004). *Building Product Populations with Software Components*. s.n.

Copyright
Other than for strictly personal use, it is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license (like Creative Commons).

The publication may also be distributed here under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license. More information can be found on the University of Groningen website: https://www.rug.nl/library/open-access/self-archiving-pure/taverne-amendment.

Take-down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from the University of Groningen/UMCG research database (Pure): http://www.rug.nl/research/portal. For technical reasons the number of authors shown on this cover page is limited to 10 maximum.

Download date: 30-05-2023

Chapter 1

Introduction

1.1 The Problem

This thesis studies the creation of software embedded in consumer products such as televisions. The study follows two empirical observations [13], both a consequence of Moore's law [62]:

- Embedded software in consumer products doubles in size every two years.
- *All* consumer products will eventually embed software and follow the same growth curve; only the starting point in time may be different.

As a result, consumer electronics manufacturers such as Philips face the following three challenges:

- How do we ensure that the *quality* of the embedded software remains high while the size and complexity of the software is growing?
- How can we build a large *diversity* of products without unnecessary duplication of effort?
- How can we decrease *time to market* to make sure that we are still (one of) the first to introduce new products?

Ongoing miniaturization and subsequent cost reduction induce a fourth challenge, that of *convergence* between thus far different products:

- Can we *combine* the functionality of existing but different products to create interesting new products, without having to re-implement the software?

This thesis seeks answers to these four challenges. In Chapter 1, we examine the business perspective and the technology trends, leading to our research questions. Chapters 2-8 contain the results of our research in the form of papers published at conferences or in journals. Chapter 9 discusses these results in terms of the original research questions.

1.2 A Business Perspective

Philips televisions have embedded software since 1978. The amount of software is growing roughly with a factor of two every two years. Figure 1 illustrates this graphically for high-end televisions. This growth is not specific for high-end TVs; it also holds for low-end TVs, for video recorders, DVD and CD players, and it is likely to hold for (mobile) phones, digital cameras, washing machines, and shavers as well. The only difference is a shift in time of the growth curve.

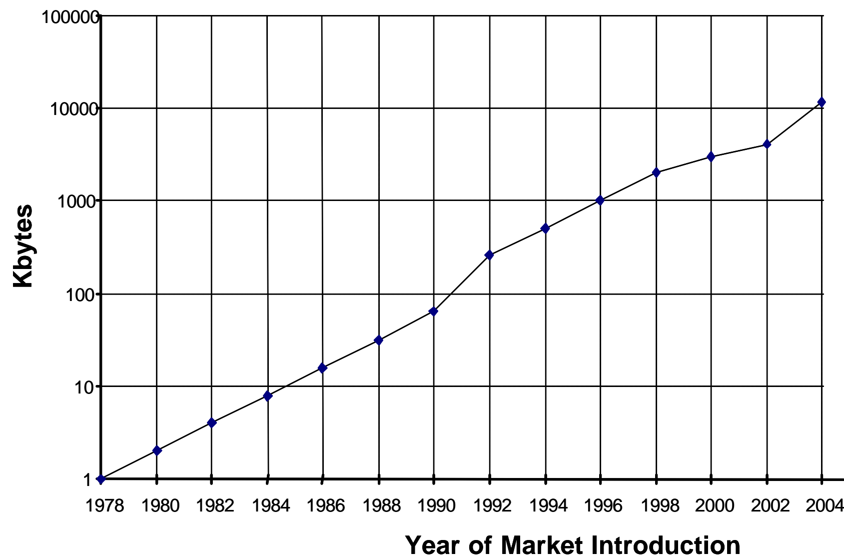

Figure 1. Size of embedded software in high-end televisions.

We expect that every consumer product manufacturer will eventually face the problem of having to build an ever-growing amount of software, and thus becoming – at least partially – a software company [95]. We believe the television is the forerunner in this: a TV with embedded microcontroller entered many households *before* the PC. For that reason, we took the TV as subject of our study.

The first challenge is that with size, complexity grows. Development that starts as a single person activity soon grows into a team effort. Software for current TVs requires more than a hundred developers and more than two years to develop. Clearly, to bring this to a good ending, there must at least be a good architecture and an efficient development process.

But this is not all. Each of the products mentioned above (TV, DVD, mobile phone, washing machine) is not alone, but part of a *family* of strongly related products, differing in price, featuring, supported standards, cultural preferences, and other criteria. Open any catalog of any manufacturer, and you will find at least a dozen different products from which you can choose. Through such a product portfolio, the manufacturer aims to obtain a large market share. The technical consequence of

this is that the software architecture and development process must be such that variants of the product can be made with little effort and as quickly as possible.

The product portfolio is not static but changes over time, setting or following the market trends. This means this it must also be possible to create *new* versions of the product in a short time. As explained above, the initial development of the software for a new television costs more than two years. Commercially, a new range of televisions must come out every year, synchronized with Christmas shopping and major annual sports events. As a more recent trend, this *time to market* must still be decreased: if new products do not come out every 6 months, manufacturers loose shelf space in the shops.

The fourth challenge is that miniaturization and cost reduction enable the creation of integrated products that combine functionality of thus far separate products. A striking example in 2003 was the printer/scanner/copier, of 2004 the mobile phone with built-in digital camera. The original example that was the direct cause of our study was the TV-VCR combination, now largely replaced by the TV-DVD combination. Note that the combination provides functionality beyond that of the parts: e.g. for the phone, taking a picture and sending it to friends or family.

Creating 'combi' products is difficult for two reasons:

- It is hard to predict which combinations will be successful beforehand.
- Development of the combined product crosses organizational boundaries.

These effectively rule out the creation of an overall architecture from which the separate and combined products can be derived. The efficient creation of a *product population*, a set of product families from which combination products can be derived, is the main topic of this thesis.

1.3 Technology Trends

This thesis builds on three main trends in software engineering in the past decade:

- Software *architecture* is increasingly recognized as an essential ingredient for building large and complex systems.
- The 'old' idea of reusable software *components* is revived through the creation and successful introduction of several component technologies.
- Software *product lines* now receive systematic attention.

Brooks already mentions *architecture* in 1975 as an important element for building large systems [18]. However, it is not until the late 1980s that software architecture becomes a discipline of its own. Shaw [97] pleads for higher-level abstractions than programming languages and abstract data types to build larger scale systems. Schwanke, Altucher and Platoff [96] define architecture as the set of allowed connections between software units, and propose ways to automatically verify an

implementation against an architecture. Perry and Wolf [86] propose a foundation for the study of software architecture, and name multiple views and architectural style as important elements. In 1995, Soni, Nord and Hofmeister propose four architecture views [101], while Kruchten proposes 4+1 views [47]. A special issue on software architecture of the IEEE Transactions on Software Engineering [40], and a book on software architecture by Shaw and Garlan [98] are a further landmark in the acceptance of software architecture as an important discipline.

The idea of reusable *components* is as old as 1968, when McIlroy advocated mass produced software components [54]. However, the *essence* of components: a deployment independent of other components and of the systems in which the components are being used, was not realized until the 1990s, except for special cases such as drivers for an operating system and mathematical and graphical libraries. In 1990, Microsoft's COM started as a foundation for creating compound documents (OLE) [113], and was soon applied to automating programs (DDE) and to implement Visual Basic controls (later called ActiveX) [17]. Around 1996, OMG defined the CORBA Component Model to build large distributed applications. Sun invented JavaBeans as building blocks for Java [42] programs. There is now an active component based software engineering community, and a recent focus of attention is the study of non-functional properties of systems built from components.

Where the component community generally takes a bottom-up view on system construction, the *product line* community essentially takes an integral view on the building of families of products. Already in 1972, Parnas studied the creation of families of programs [85] (spending one year in Philips at the time). Non-software product lines are almost as old as factories; software product lines have also been around for quite some time, CelsiusTech (formerly Philips) being a famous example [21]. Around 1995, the Software Engineering Institute started a program on software product lines [100], the ARES project studied product families [3], and major conferences held sessions on product lines [87]. Most researchers emphasize the analysis of commonality and variability at an early stage, to build systems with a sufficient number of variation points to implement different products; as a result, the product line and component communities are quite disjoint.

Other fields have a relation with our work too, most notably *partial evaluation* and *configuration management*, but we shall not discuss them here. Their role will become evident in the following chapters.

To relate our problem statement to the trends described in this section: we want to build *high quality* software for a *family* of consumer products in a *short time*, and we require an approach that can be extended to build product *populations*, a set of product families from which we can derive 'combi' products. Architecture will help us to manage the complexity and thus the quality of the software, while reusable components (bottom-up) and software product lines (top-down) will help

us to efficiently build families and populations. Figure 2, adapted from [81], illustrates this graphically.

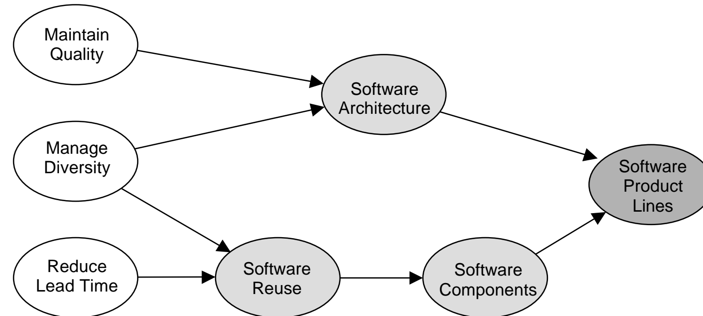

Figure 2. Main drivers for software product lines

1.4 Research Questions

An architecture will only help to improve the quality of a system if that architecture is *explicit* so that it can be analyzed (or at least discussed), and if it *consistently* describes the implementation at some level of abstraction. Much of our earlier work deals with formalizing architecture and verifying implementations against it.

The first in this field was Warshall [110], using an efficient algorithm to calculate the transitive closure of a binary relation. Schwanke, Altucher and Platoff [96] extracted architecture from code and represented it as graphs, so that they could compare the *actual* architecture against the *intended*. They also define aggregation (part-of) relations so that they could inspect the use relation at higher levels. We pursued a similar route in the early 1990s [68][26], using relation algebra to provide a mathematical basis [27]. Holt [35] followed essentially the same approach. Our work finally culminated in an experience paper [15] that shows that verification of architecture *can* be made successful but not without continuous attention. A more promising way is to use an architectural description language in *forward* mode, i.e. to create explicit descriptions of architecture and generate code from that. Darwin [52] is an example of such a language.

To build product families and populations, we need a powerful mechanism to deal with commonality and diversity, and we believe – with many others - that software components provide a solution. A closer investigation reveals two complementary ways to use components for this, as illustrated in Figure 3. One way is to build a product independent framework in which product specific components can be plugged. Another way is to build product independent components that can be combined in product dependent ways. The first way we name *variation*, the second way *composition*. Many researchers in the product line community favor variation, while the component community favors composition. Variation may be good to build families, but we believe that composition is needed to build populations.

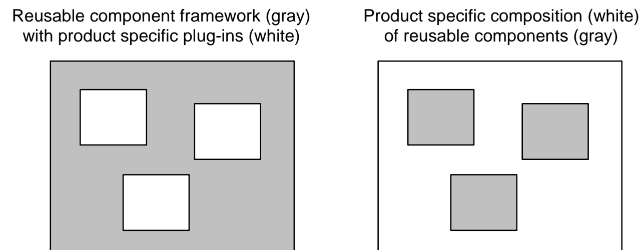

Figure 3. Variation (left) and composition (right) are complementary approaches.

When we started this work in 1996, the available component models were Microsoft's COM, CORBA's component model and JavaBeans. The latter was tied to a Java virtual machine, which we did not have in a television. CORBA was deemed unsuitable for an embedded system, so COM was the only option. Especially the *QueryInterface* concept of COM was useful, as it allows to implement new interfaces while still supporting the old ones, a feature necessary for evolution. The downside of COM is that VTables provide code size and run-time overhead.

So we had two choices: wait 5 years (estimated) for the computing power of a TV to be such that COM can be used, and bridge the gap with other techniques for handling diversity, or adjust the technology so that it fits in a TV, and start gaining experience with component based design. We chose the latter as we believed that components were the future, and we wanted to tune our software development process and organization to component based design as soon as possible. The resulting component technology is called Koala [70].

Summarized, here are the four research questions that we formulated in 1996, and that are the topic of this thesis.

- Can *software architectures* be made more *explicit*, and does this increase the quality of products?
- Can *component technology* help to build product *families* and *populations*, and what design patterns are needed for that?
- Can component technology be applied in *resource-constrained products*, and what consequences does this have on the technology?
- Can this all be made into a business success, and what impact does this have on development *process* and *organization*?

We believe that each of the questions above warrants a study on itself. The value of this thesis is not as much in answering the questions individually and in depth, but rather in combination and integration. We present a method that addresses the four questions above, and that is being successfully applied at Philips at a large scale.

1.5 Way of Working

Research in software engineering differs from research in other fields of computer science in at least two aspects:

- *Scale* is an important source of complexity, making it difficult to conduct realistic (but necessarily small-scale) experiments in the laboratory.
- *Validation* of results is difficult as many factors are involved in the success or failure of a software project.

With respect to scale, Basili distinguishes researchers from practitioners [7]. The researcher tries to *understand* the nature of processes and methods to build systems, while the practitioner *builds* improved systems with this knowledge. The one cannot live without the other: the researcher needs a laboratory to observe and manipulate variables, but these only exist where practitioners build systems. Conversely, the practitioner needs to better understand how to build systems, and the researcher can provide models to help.

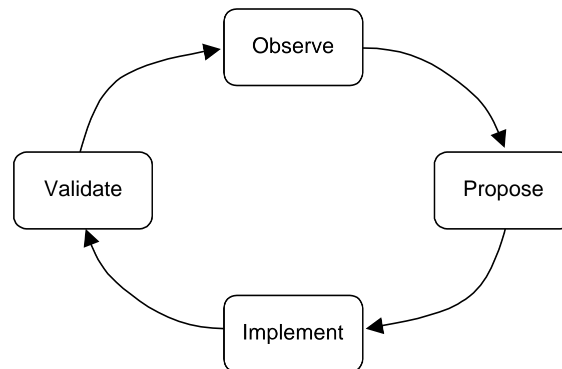

Figure 4. The empirical cycle.

For this reason, we adopted an *industry as laboratory* approach in our software engineering research [65]. We seek a close contact with a development group that builds (large) systems, and then follow an experimental paradigm (Figure 4, see also [7]):

- We *observe* the way that the group is developing software, and – in close cooperation with people from that group – identify problems in their way of working.
- We *propose* – again in close cooperation with people from that group – as hypothesis a set of solutions, and try these out ‘in the small’ at research.
- We then *implement* these solutions in the development group, and attempt to measure and analyze how well the solution is working in practice.

- This way, we can *validate* our hypothesis, and use the information to repeat our experimental cycle, thus improving our knowledge of both the problem and the solution domain.

As a research group, we have used this paradigm with many development groups and often multiple times with the same development group. This thesis describes the third cycle that we had with the group developing software for televisions. The first two cycles had the following questions as topic:

- Can *formal specifications* and *rapid prototyping* help to improve the quality of software in televisions? The answer was yes, though formal specifications turned out to be difficult to transfer into industry [43].
- Can *architecture* help to manage the increasing complexity and size of the software in televisions? The answer was positive, but specific techniques to handle diversity were needed.

The third cycle started with an explicit question from the development group for techniques to handle diversity; components and explicit architectural descriptions were our answer. We anticipated – and later noticed – that process and organization had to change as well to introduce these techniques properly.

Validation is the other difficult aspect of software engineering research mentioned above. The problem is that the success of any software development project is determined by many variables, among which a number of human factors. It is very difficult to conduct systematic research to the effects of these variables in large-scale projects; it certainly is not feasible to repeat an experiment with other values for a selected set of variables. In fact, in a software business that grows according to Moore’s law, *any* new project is bigger than *all* projects before, and therefore a first.

One of the techniques mentioned in [7] is a project-based study, as opposed to a human factors based study, where through variation of teams (*who* is building) and variation of projects (*what* is being built), conclusions can be drawn. As it happens, the method described in this thesis is being applied by over a dozen different teams that operate relatively independently, at different places in the world (Europe, US, Asia). In addition, the method has been applied to three different hardware chassis and two different market segments, each with their own characteristics. As such, our research is more of a *field study* than a *case study*.

Although we really need a *quantitative* validation of the method, a good first step is to have a *qualitative* validation. Quantitative studies are objective and oriented towards verification [7], but even though qualitative studies are subjective, they can help to discover important issues. As a qualitative validation, we can ask the following questions:

- Is the method that we devised *transferable* to industry (a necessary but not sufficient condition for success)?

- Is the development team that uses the method *successful* in building a product line (again, necessary but not sufficient)?
- Are the developers *enthusiastic* about the method, and do they feel that it has helped them to solve their problems?
- Does management believe that the method is instrumental to their success in the future?

A method should be wider applicable than one organization and one domain:

- Can the method be used by other organizations or companies, and/or in other sub domains or domains?

This requires a significant amount of effort to investigate it. A simpler question (again a necessary but not sufficient condition) is:

- Can you identify other teams that experience problems (and are aware of them) that can be solved by the method, according to you and/or the team?

The final qualitative questions (again, none of them conclusive) position the work in software engineering research:

- Are papers on the method accepted at conferences and in journals?
- Do others researchers refer to these, and do they build upon the work?

There are also quantitative measurements that we can perform. We can count the number of products and components and calculate how components are reused over products. We can measure the stability of the code base, and see whether components are reused 'as is' or continuously being modified. We can measure the stability of the interfaces, and determine whether interfaces indeed form stable points in the evolution of the system. All these numbers provide insight, but they would really gain in value when comparing them to other approaches in other companies. Unfortunately, that is outside the scope of this thesis.

1.6 Time Line

Preamble to this study were the following:

- We studied the use of formal specification techniques and rapid prototyping to increase the quality of software in televisions in 1988-1993 [43][69].
- We studied the formalization, visualization and verification of architecture in 1992-1996 (see Chapter 2 for a summary). This work was continued by colleagues until 2002.

The work described in this thesis started mid 1996. The Koala component model was defined in the period September 1996 to March 1997. The Koala compiler was implemented in the period January 1997 to June 1997. The original plan was to

apply Koala to the development of television software in the summer of 1997. Due to other circumstances, this was cancelled, and research received the assignment of designing a new component-based software architecture for televisions in 1998. This architecture, baptized MG-R, created its first product in 2000. By 2002, *all* mid-range and high-end television products were based on MG-R.

The origin of the name Koala is depicted in Figure 5. COLD, IGLOO, ICE and ICE BEAR were a language, a library, a development environment and a graphical user interface respectively for writing formal specifications. Polar and Panda were a graphical language and a graphical editor for COLD. Teddy and Ursa are tools for architecture visualization and verification. Darwin is an ADL for specifying distributed systems, and Kangaroo is an ADL for describing user interfaces.

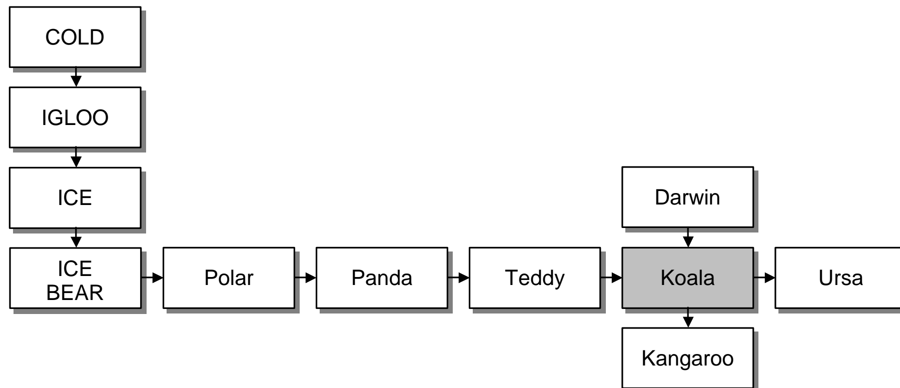

Figure 5. Derivation of the name Koala

1.7 Overview of this Thesis

The main body of this thesis consists of articles that have been published in major journals (chapters 2, 3, 8), conferences (chapters 4, 5, 7) and workshops (chapter 6). The papers have been left 'as is', with the exception of some minor error corrections and typographical changes. The articles are provided mostly in chronological order, but where necessary, we deviate from this order to enhance the logical flow of the thesis.

For a quick overview of Koala, read Chapter 3. For a quick overview of the use of Koala to build software for televisions, including process and organization issues, read Chapter 7.

Chapter 2, published in *Computer Languages* in 2001 [79], describes the work on the formalization, visualization and verification of architecture that we did prior to the work on Koala. It shows how familiar concepts in architecture (layers, subsystems) can be provided with a mathematical basis, and how then tools can be built to visualize such structures and to verify whether implementations actually conform to these structures. The conclusions from this work were twofold:

- It is very useful to have a formal notion of architecture, especially if there is a graphical notation accompanying this.
- It is possible to verify an implementation against an architecture ‘after the fact’, but it is then often too late to correct faults in either implementation or architecture.

This work therefore led to our first research question: can we make architecture explicit beforehand?

Chapter 3, published in *IEEE Computer* in 2000 [70], postulates complexity and diversity as the main issues in the development of software for televisions, and proposes an explicit description of architecture and the (re-)use of components as solution. Koala is then described in two steps: first the basic model (components, interfaces, modules, binding) and then the extended model (function binding, diversity interfaces, switches and partial evaluation).

Chapter 4, published at the *Working IEEE/IFIP Conference on Software Architecture* (WICSA) in 2001 [77], discusses independent deployment as the bare essence of software components. This paper was the direct result of many discussions within Philips, where many development groups claimed to already use components, but in a decomposition paradigm, so in a dependent deployment. The paper introduces a two-component system consisting of a client and a server, where each component can vary in two dimensions: time and space. The resulting four steps are discussed in detail, with ample examples. This material forms the basis of the design of Koala, although we never made this explicit until writing this paper.

Chapter 5, published at the *Second Software Product Line Conference* (SPLC-2) in 2002 [81], explains a topic left open in Chapter 3: why composition is the best paradigm to build consumer products, rather than variation. The answer lies in the intended scope of the software product line. Many familiar examples of variation and composition are given.

Chapter 6, published at the *Tenth International Workshop on Software Configuration Management* (SCM-10) in 2001 [76], compares the use of an ADL to handle diversity with the more traditional approach of configuration management, and clearly separates issues to be handled in the ADL domain from issues to be handled by configuration management tools.

Chapter 7, published at the *International Conference on Software Engineering* (ICSE) in 2002 [80], contains a full overview of our approach, and as such, it is representative for – and has the same title as – this thesis. The paper is structured in terms of Business, Architecture, Process and Organization aspects; one of the important lessons that we learned is that these cannot be treated in isolation. Put more strongly: a component-based architecture is only a success if it serves a clear business goal and it is accompanied by a matching development process and organization.

Chapter 8, published in *Software Practice and Experience* in 2003 [83], answers an important question in the design of (low-level) control software for televisions: how to make the control software ‘composable’. It demonstrates an architectural style of controllers that communicate horizontally to guard the integrity of the signal path. An efficient implementation technique of direct function calls makes this work in a resource-constrained environment.

Finally, Chapter 9 provides a validation of the work reported in this thesis.