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 5

From Variation to Composition

Published as: *Widening the Scope of Software Product Lines – from Variation to Composition*, Rob van Ommering, Jan Bosch, Second Software Product Line Conference, San Diego, August 2002, LNCS 2379, p328-347.

Abstract: Architecture, components and reuse form the key elements to build a large variety of complex, high-quality products with a short lead-time. But the balance between an architecture-driven and a component-driven approach is influenced by the scope of the product line and the characteristics of the development organization. This paper discusses this balance and claims that a paradigm shift from variation to composition is necessary to cope with an increasing diversity of products created by an ever-larger part of an organization. We illustrate our claim with various examples.

5.1 Introduction

As a software engineering community, we have a long history in building software products: we have been doing this for half a century now. *Architecture* has always been an important success factor [18], though it has not received the attention that it deserves until the last decade. *Components* (or modules) serve as a means to cope with complexity (divide and conquer), or to streamline the evolution of systems [85]. And we are already *reusing* software for a long time: operating systems, graphical and mathematical libraries, databases and compilers are hardly ever written from scratch for new products.

It is therefore not surprising that people developed the vision that the future of software engineering lies in the use of reusable components: new applications would be built by selecting from a set of existing components and then clicking or gluing them together. McIlroy was one of the first to advocate this idea [54], and many have followed since. We have indeed seen some successes in this area, for instance the use of ActiveX controls (components) and Visual Basic (glue) to quickly build interactive applications for PCs. But we have also seen many failures, especially when applied *within* companies to create a diversity of products.

© 2002 Springer Verlag. Reprinted, with permission, from Software Product Lines, Proceedings of the Second International Conference, SPLC 2, San Diego, CA, USA, August 19-22, 2002, Rob van Ommering, Jan Bosch, 'Widening the Scope of Software Product Lines – from Variation to Composition', Springer Verlag LNCS 2379, p328-347.

This is where software product lines enter: proactive and systematic approaches for the development of software to create a variety of products. Again, software product lines have been around for decades, but only recently receive the attention that they deserve. Note that product lines are not specific to software only: they are also commonplace in for instance the car, the airplane and the food industry [21].

Most software product line research focuses on an early analysis of commonality and variation [5], followed by the creation of a variant-free architecture [88] with a sufficient number of variation points [41] to deal with diversity. While this works well for small product families in a small organization, we have experienced problems when applying this to larger ranges of products, especially when development crosses organizational boundaries.

This paper is about widening the scope of product lines to encompass a larger diversity of products (a *product population* [73]), where the development takes place in different organizational units but still within the same company. We believe that the successful building of product populations is the next hurdle to take in software engineering. Figure 29 illustrates this, showing on the one hand that the focus of our community shifts over time from products to components to families to populations, and on the other hand that we are seeking a balance between variation (as in product families) and composition (as in software components).

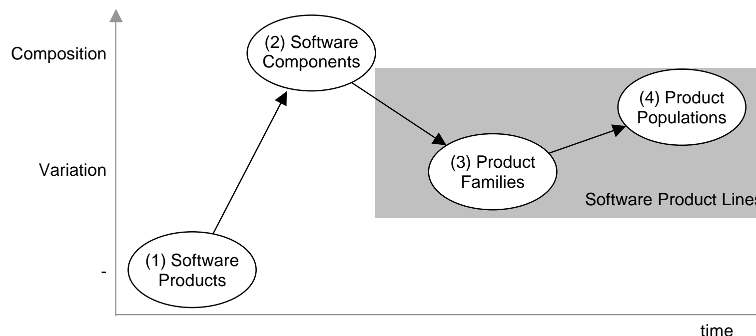

Figure 29. Towards product populations

This paper is organized as follows. First we recapitulate the main drivers for software product lines. Then we discuss the influence of product scope and characteristics of the organization on software product lines. We proceed with introducing variation and composition as separate dimensions, and then explain variation and composition in more detail. We end with some concluding remarks. We take examples from the domain of one of the authors, the control software for consumer electronics products, but include examples from other domains as well.

5.2 Why Use Software Product Lines?

Let us first summarize *why* we want to create software product lines. In Figure 30, we show the main four drivers as we identified them within Philips at the left hand side. We show three key solutions in the middle. But it's the combination of these three that forms the most promising solution to our problems: software product lines.

In the next subsections, we discuss each of the drivers and their relation to the solutions. Please note that arrows in Figure 30 show main dependencies only; arguments can be found for other dependencies as well, but adding them would obscure the main flow of the picture.

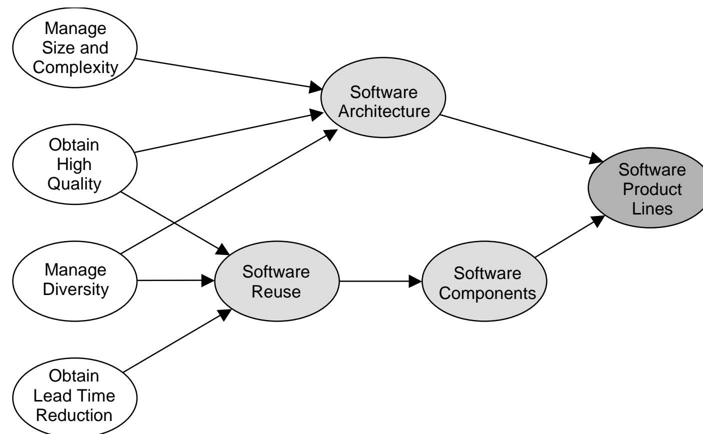

Figure 30. Main drivers for software product lines

5.2.1 Manage Size and Complexity

Software plays a role of ever-increasing importance in many products of today. In Philips televisions, software was introduced in the late 70s, when a 1-kiloByte 8-bit microprocessor was first used to control the hardware signal processing devices. The size of the software grew following Moore's law closely: today, a TV typically has 2 MegaBytes of software. The same trend can be observed in other products, though the point in time where software starts to play a role differs. Typically, televisions run 10 years behind on PCs, while shavers run 10 years behind on televisions.

The best – perhaps only – way to manage size and complexity is to have a proper architecture. A software architecture defines, among other things, a decomposition of a system in terms of components or modules [39]. One common misconception (at least in our surroundings) is the assumption that components defined to manage size and complexity, automatically help to satisfy other drivers as listed in Figure 30 as well, most notably diversity. Experience has shown this not to be the case: defining components to manage diversity is an art per se.

5.2.2 Obtain High Quality

The second driver in Figure 30 is quality: the more complex systems become, the more difficult it is to maintain a high quality. Yet for systems such as televisions, high quality of the software is imperative[4]. Users of a television do not think they are communicating with a computer, so they will not accept crashes. Also, although lives may not seem to be involved, a TV catching fire can have disastrous effects, and it is partly the software that prevents this from happening.

Quality can be achieved by having a proper architecture that guarantees (or at least helps to satisfy) certain quality attributes. Quality can also be achieved by reusing software that has already been tested in the field.

5.2.3 Manage Diversity

The third driver in Figure 30 is the first one specific for product lines: managing the efficient creation of a variety of products. For consumer products, we have 'small-scale' diversity within a family of televisions: the output device, the price setting, the region, the user interface, the supported standards, interconnectivity et cetera. We have 'large-scale' diversity if we want televisions, VCRs, DVD/CD players/recorders, etc. to be part of one product line approach. There is indeed sufficient commonality to justify the sharing of software between those products. We use the term *product population* to denote the large-scale diversity. Other examples of product populations can be found in the medical domain [50] and for printers and scanners [9].

There are actually two schools of thought for managing diversity, which we believe are two extremes in a spectrum of solutions. One is to build an architecture that expresses the commonality between products, with variation points to express the diversity. The other is to rely on reuse of software components for the commonality, but to use different compositions of these components to implement diversity. This paper is about balancing between these two solutions, variation and composition, depending on the scope of the products and the characteristics of the organization.

5.2.4 Lead-time Reduction

The fourth driver in Figure 30 is lead-time reduction. Although in principle also relevant for single product development, we think that it is especially relevant in product line contexts, since the goal of product lines is to produce a range of products fast. The main solution to obtain lead-time reduction is reuse of software. Experience has shown that higher-level languages, modern development methods,

[4] Repairing software shipped in televisions sold to a million customers is as difficult as in a rocket sent a million miles into space.

better architectures and advanced software development environments have indeed improved software productivity, but not with a factor that is sufficient to obtain the speed required in product lines.

5.2.5 Does Software Reuse imply Software Components?

Figure 30 may suggest that software reuse implies the deployment of reusable software components. Indeed, reuse of software components is one (compositional) approach to build product populations, and one that we will examine in this paper. But we should note that we can also have reuse *without* software components, for instance when using inheritance in an object-oriented framework. And, perhaps surprisingly, we can also build a product line that utilizes software component technology without actually reusing the components! An example of the latter is a framework with component plug-ins, where the plug-in components implement *diversity* rather than *commonality*. Figure 31 illustrates this: the left hand side shows a reusable framework with product-specific plug-ins, while the right hand side shows reusable components, composed in a product-specific way.

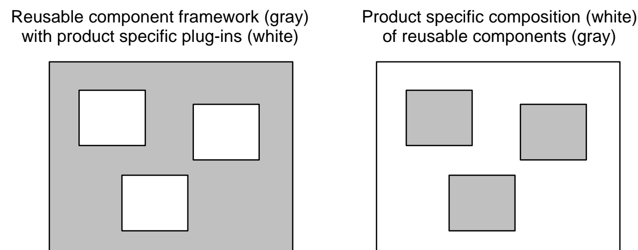

Figure 31. Variation (left) and composition (right) are complementary approaches

5.2.6 Finding the Balance

In the rest of this paper, we concentrate on the balance between the use of variation and composition in software product lines, depending on the scope of the product population and the characteristics of the organization building it.

5.3 The Influence of Scope on Software Product Lines

As we have said before, many software product line approaches advocate an early analysis of commonality and variation of the products to be built, resulting in a variant-free architecture with variation points. This is indeed a valid approach, but there may be circumstances in which the preconditions for such an approach are not satisfied. We shall discuss two categories of these in the following sections: (1) product related and (2) organization related.

5.3.1 From Product Family to Product Population

Traditional product lines cover families in which products have many commonalities and few differences. This enables the creation of a variant-free architecture with variation points to implement the differences. The variation points typically control the internals of a component, but they may also control the presence or absence of certain components, the selection of a concrete component to implement an abstract component (a place holder), or even the binding between components.

A typical product family in the consumer domain is the set of televisions (direct view, flat screen, projection) produced by a single company[5]. Let us now extend the product range to include video recorders, CD and DVD players, CD, DVD and hard disk recorders, audio amplifiers, MP3 players, tuners, cassette players, et cetera. These products still have sufficiently in common to warrant reuse of software. But they are also sufficiently different to require different architectures.

The latter is actually not a fundamental problem. In principle, it seems perfectly possible to create a single architecture for a 'super consumer product', being the combination of all products mentioned above, and to strip this architecture for any subset to be sold. While this is certainly possible in hindsight, it is very difficult to predict which combinations of audio and video elements will actually be interesting next year, and to include that already in the architecture.

Instead, the market itself thrives on novel combinations of existing functions (for instance a mobile telephone with built-in digital photo camera). We believe that in such an evolving 'compositional' market, the software architectures must also be compositional. Put differently, variation can be used to implement diversity known a priori, while composition can be used for open-ended diversity.

5.3.2 From Business Line to Product Division and Beyond

The second argument in favor of composition has its roots in organization. A large company like Philips is typically organized in product divisions (Semiconductors, Consumer Electronics), which are split into business units (video, audio), which again are split into business lines (Flat TV, projection TV). See Figure 32 for an illustration.

Within a business line, setting up a software product line is generally relatively easy. One person is in charge of the business, which has its own profit and loss calculation, and a product line is demonstrably an efficient way to create a variety of products. The products are reasonably alike (e.g. all high-end Flat TVs). Moreover, the product road maps in a business line are usually carefully aligned,

[5] For technical reasons, low-end televisions usually belong to a different family than high-end televisions.

thus enabling the creation and use of shared software. A single architecture with variation points is very feasible.

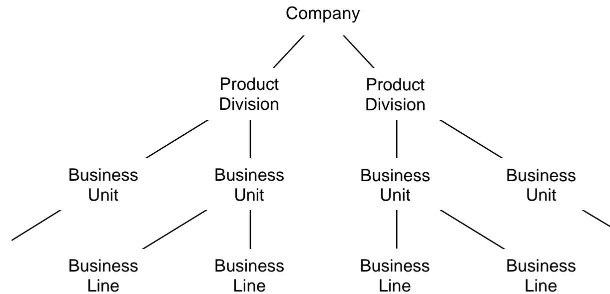

Figure 32. Schematic organization of a large company

Compare this with a business unit. Although – of course – still one person is in charge, profit and loss responsibility is delegated to the business lines, making sharing software between business lines at least cumbersome. Each business line manager has his own agenda for planning software development. Protection of IP (intellectual property) is sometimes an issue, but more frequently the misalignment of roadmaps causes problems: one business line is changing software with its product release still far away, while another product line needs a mature version of the software for its immediate next release. A careful and explicit road mapping is required here [78]. Products show more differences, but in general a single architecture with variation points is still be feasible.

Problems get worse at the level of a product division. Not only the products become more different (e.g. TV versus set-top box), the managers more apt to look after themselves instead of after their common interest, and the roadmaps harder to align, but also the cultures between organizations become different. In Philips, the TV business is a traditional and stable consumer business (low profit per product but high number of products, targeted directly at the person in the street), while the set-top box business is new and evolving (currently targeting at large providers who deliver the boxes free with their services). TVs are typically programmed in C in a few MegaBytes, while set-top boxes are programmed in C^{++} in many MegaBytes.

Finally, product divisions act as independent companies (at least in Philips), and while sharing of software between them is in many cases infeasible (as between medical systems and consumer electronics), it may in certain situations still be desirable. Philips Semiconductors produces for instance many of the chips for Philips Consumer Electronics, and it also sells the same chips to third parties, so certain software can be better leveraged if developed by the semiconductors division. This puts extra demands on protection of IP. Cultures between product divisions are even more different, and the levels of software engineering maturity may differ.

5.3.3 How to Share Software in Large Organizations?

It should be obvious by now that the top-down creation of a single architecture with variation points to cover all consumer products within Philips is infeasible. Instead, we should find a way to let different business lines create the software components in which they are specialized (e.g. the business line DVD creates a DVD unit), and find ways to ‘arbitrarily’ compose them.

Should we then strive for a bottom-up approach as advocated by many believers in software component markets, i.e. (2) instead of (4) in Figure 29? The answer is no: we still need a careful balance between top-down and bottom-up, as we need to make sure that components fit easily, as explained in the next section.

Component-driven
Bottom-up
Opportunistic
Available
Inter-organization
Architecture-driven
Top-down
Planned
To be developed
Intra-organization
Extreme Programming
Design With Reuse
Domain Architectures
Roadmapping
Sub contracting
Design For Reuse
Product Populations

Figure 33. Balancing between architecture- and component-driven

5.3.4 Balancing Architecture- and Component-Driven

Classical product and product line approaches ((1) and (3) in Figure 29) are architecture-driven, top-down, planned, intra-organization, with all software to be developed in the project. Component market evangelists ((2) in Figure 29) on the other hand, claim that software development should be component-driven, bottom-up, opportunistic, inter-organization and using components that are already available. Figure 33 shows both sets of in fact opposite characteristics along different axes. Note that they are not all dependent – we can make cross combinations:

- *Extreme programming* could be seen as an opportunistic bottom-up yet architecture-driven method.
- Top-down, architecture-driven development deploying available software is sometimes called *design with reuse*, while on the other hand component-driven bottom-up development is sometimes called *design for reuse*.
- A top-down, architecture-driven development spanning multiple organizations is possible but only if there is consensus on a *domain architecture*.

- The planned development across organizations of new software is often called *subcontracting*, and should not be confused with the development of reusable software[6].

Our vision is to achieve a component-driven, bottom-up, partly opportunistic software development using as much as possible available software to create products within our organization; we call this a *product population* approach. In reality, life is not so black and white. More specifically, we believe that such an approach should be:

- Component-driven (bottom-up) with a lightweight (top-down) architecture, since absence of the latter would result in architectural mismatches [33].
- Opportunistic in using available software but also planned when developing new software, the latter carefully roadmapped [78].
- Intra-organization but spanning a large part of the organization.

In a way, building product populations is more difficult than producing and utilizing software components on a component market. In the latter case, laws of economics apply: component vendors can leverage development costs over many customers (as compared to few in product populations), while component buyers can find second sources in case vendors does not live up to expectations. One would expect that the same economical model could be used *within* an organization, but in practice this does not work: contracts within a company are always less binding than between companies.

5.3.5 Third Party Component Markets

Does this mean that we not believe in third party component markets [109] then? Yes, we do, but not as the only solution.

First of all, most software components on the market tend to implement generic rather than domain specific functionality. While this ensures a large customer base, it does not help companies like Philips to build consumer products. Put differently: of course we reuse generic components such as a real-time kernel from third parties, but there is still a lot of software to be created ourselves. This situation may change with the arrival and general acceptance of TV middleware stacks.

Secondly, many companies distinguish between *core*, *key* and *base* software. Base software can and should be bought on the market. Key software can be bought but is made by the company itself for strategic reasons. Core software cannot be bought because the company is the only company – or one of the very few –

[6] Within Philips, we have seen a single team that had as mission the development of reusable software act in practice as two teams subcontracted for two different product projects.

actually capable of creating such software. But there may still be a need to share this software within the company!

Thirdly, many components out on the market are still relatively small (buttons, sliders), while the actual need for reuse is for large components. Of course there are exceptions: underlying Microsoft's Internet Explorer is a powerful reusable HTML displaying *and* editing engine that can be used in different contexts (such as Front Page and HTML Help). But such examples are still exceptions rather than a rule.

Finally, to be accepted in a market, functionality must be mature, so that there is consensus in the world on how to implement and use such functionality. This process may take well over 20 years – consider operating systems and windowing systems as examples.

5.4 The Dimensions of Variation and Composition

Figure 34 shows variation and composition as two independent dimensions. The variation axis is marked with 'As is' (no variation), parameterization, inheritance and plug-ins, as subsequently more advanced techniques of variation. The composition axis is marked with no composition (yet reuse), and composition. The table is filled with some characteristic examples, which we briefly discuss in the subsections following the table. In the sections following, we zoom into variation and composition with more elaborate examples.

		COMPOSITION →	
		Reusable but not composable	Composable
VARIATION ↓	'As is'	Libraries	LEGO
	Parameterization		Visual Basic
	Inheritance	OO Frameworks	Frameworks as Components
	Plug-ins	Component Frameworks	

Figure 34. Variation and composition as two independent dimensions

5.4.1 Plain 'Old-Fashioned' Libraries

The classical example of software reuse is through libraries. Examples are a mathematical library, a graphical library, a string manipulation library, et cetera. The functions in the library are either used 'as is', or they can be fine-tuned for their specific application using a (procedural) parameter mechanism.

Typically, software deploying the library includes the header file(s) of the library, and thus becomes specifically dependent upon the library. It is possible to abstract

from the library by defining a library-independent interface; POSIX is an example. The build process may then select the actual library implementing this interface. However, in practice, most software built on top of libraries is still specifically dependent upon those libraries.

This is especially true for the library itself. Often, a library uses other libraries, but few of them allow the user to select which underlying libraries to use. Moreover, while some libraries are relatively safe to use (they define specific functions only such as 'sin' and 'cos'), other libraries may introduce clashing names (String), or have clashing underlying assumptions (both define the function 'main').

The conclusion is that while libraries are a perfect low-level technical mechanism, more advanced techniques are necessary to achieve the variation and composition required for product lines.

5.4.2 Object-Oriented Frameworks

An object-oriented framework is a coherent set of classes from which one can create applications by specializing the classes using implementation inheritance. An example is the Microsoft Foundation Classes framework [92]. The power lies in the use of inheritance as variation mechanism, allowing the framework to abstract from specific behavior, to be implemented by the derived classes. A base class usually has a very intricate coupling with its derived classes, the base class calling functions implemented in the derived class and vice versa.

This intricate coupling is also one of the weaknesses of this approach. A new version of the framework, combined with old versions of the derived classes may fail due to subtle incompatibilities; this is known as the *fragile base class problem* (see [103] for a treatment). Also, since many frameworks are applications themselves, combination of two or more frameworks becomes virtually impossible.

5.4.3 Component Frameworks

A component framework is (part of) an application that allows to plug-in components to specialize behavior. Component frameworks resemble OO frameworks; the main difference is the more precise definition of the dependencies between the framework and the plug-in components. The plug-ins can vary from quite simple to arbitrarily complex, in the latter case enabling the creation of a large variety of applications.

Component frameworks share some of the disadvantages with OO frameworks. First of all, the plug-in components are usually framework dependent and cannot be used in isolation. Secondly, the framework often implements an entire application and can therefore not easily be defined with other frameworks.

5.4.4 LEGO

The archetype non-software example of composition is LEGO, the toy bricks with the standard 'click and play' interface. Indeed, LEGO bricks are very composable, but they support no variation at all! This results in two phenomena.

First of all, it *is* possible to create arbitrary shapes with standard LEGO bricks, but one must use thousands of bricks to make the shapes smooth (see Figure 35 for a pirate's head). As the components are then very small compared to the end product, there is effectively not much reuse (not more than reusing statements in a programming language).

To solve this, LEGO manufactures specialized bricks that can only be used for one purpose. This makes it possible to create smooth figures with only few bricks. The down side is that specialized bricks can only be used in specialized cases.

Figure 35. The pirate's head with ordinary LEGO bricks[7]

A solution could be parameterized bricks, which indeed is a mechanical challenge! Maybe a LEGO printer, a device that can spray plastic, could be an answer here![8]

5.4.5 Visual Basic

The archetype software example of composition is Visual Basic. Actually, Basic is the non-reusable glue language, where ActiveX controls are the reusable components (later versions of VB also allow to create new ActiveX controls).

One of the success factors of Visual Basic is the parameterization mechanism. An ActiveX control can have dozens (hundreds) of properties, with default values to alleviate the instantiation process. Another parameterization technique is the event

[7] From http://www.lego.com

[8] Strictly speaking, a LEGO printer would be the equivalent of a code generator, which is a technique for building product lines that we do not tackle in this paper, although it can be very fruitful in certain domains (cf. compiler generators). See also [23].

mechanism, which allows the signaling of (asynchronous) conditions, where the application can take product specific actions.

Still, ActiveX controls are usually quite small (the Internet Explorer excepted), and it is not easy to create application frameworks (e.g. supporting a document view control paradigm such as the Microsoft Foundation Classes do).

5.4.6 Frameworks as Components

The ultimate solution for variation *and* composition lies – we believe – in the use of (object-oriented or component) frameworks as components. We have seen that frameworks provide powerful variation mechanisms. If a framework only covers part of an application domain, and if frameworks can be arbitrarily combined (making them *true* components), then we can create a multitude of products by selecting and combining them and then specializing them towards a product.

There are some examples of OO frameworks used as components [11], but no examples of component frameworks used as components yet. The basic mechanism for achieving composition is to make every context dependency explicit and bindable by a third party, as we will see in a later section.

5.5 Variation Further Explained

Let us zoom into variation first, and then discuss composition in more detail.

5.5.1 Why do we Need Variation?

The basic argument why we need variation is the one underlying all product line development: we need to implement diversity. We can achieve this with reusable components, but an interesting dilemma pops up then. The more useful we make a component (read: larger), the less reusable it becomes. Conversely, the more reusable we make a component, the less useful it may become, the ultimate being the empty component: extremely reusable while not very useful! See also [103], Fig 1.1).

The solution for this dilemma is *variation* as a way to fine-tune components when instantiating them into a product. The archetype example is the Visual Basic ActiveX control, which usually comes with a long list of properties that can be modified at design- and/or run-time. These properties also have default values: this alleviates the product creation; otherwise the burden on the designer may be too high.

5.5.2 Types of Variation

A variation point can be implicit or explicit, open or closed, unbound or bound, and flexible or fixed [11]. Not all sixteen combinations are meaningful; see Figure 36 for an overview. When development starts, a variation point is implicit; it must be

made explicit during analysis and design. An explicit variation point can be open if the set of possible values can still be extended, or closed otherwise. A variation point is unbound if a value has not been selected, and bound otherwise. A variation point is fixed if it's bound and the value cannot be changed anymore.

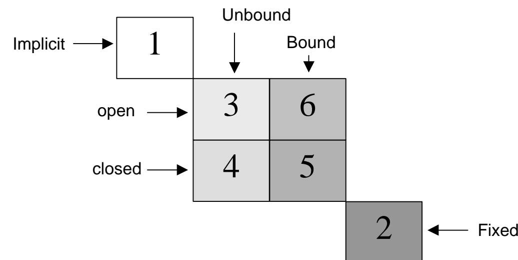

Figure 36. A lifecycle for variation points

Traditional single product development starts at (1) in Figure 36 and then immediately jumps to (2). Classical product family development proceeds through (3), (4) and (5) (identify a variation point, identify its values, and then choose a value) before ending in (2). There is a tendency to make (5) the end point of development, allowing the variation point still to change at the customer site. There is even a tendency to make (6) the end point of development, allowing the customer to extend the set of values.

5.5.3 Example 1: Switches and Options

In our previous television software architecture, we distinguished between *switches* and *options*. A switch is a compile-time variation point (an #ifdef), closed, bound and fixed somewhere during development time. An option is a run-time variation point, a value stored in non-volatile memory and used in an if-statement, so closed yet unbound during development, and bound in the factory or at the user's home.

These variation points were used for adjustments of hardware, for presence or absence of components, and rarely also for selecting between components. The set of variation points was very specific for the product family, and could not be used to achieve the large-scale structural variation that we needed to widen our family.

5.5.4 Example 2: The Windows Explorer

The windows explorer is an example of an application with many explicit, open, unbound and flexible variation points. It can be tuned into almost any browsing application, though it cannot be used as component in a larger application. In other words, it supports a lot of variation, but no composition.

The explorer can be tuned at a beginner's level and at an advanced level [59]. In the first case, simple additions to the windows registry suffice to change the file association (the application to be run for a certain type of file), the context menus

or the icons. In the second case, handlers must be registered for specific shell extensions, and these handlers must be implemented as COM components. When such a handler crashes, the explorer crashes, with all its windows including the desktop[9].

The folder view of the explorer can be modified by name space extensions; some FTP clients use this to add favorite sites to the name space. The folder view is part of the explorer bar, which can also host other views (search, favorites, history). The right pane normally hosts a standard list view control, but it can also host the Internet Explorer to view (folders as) web pages, or any other user defined window.

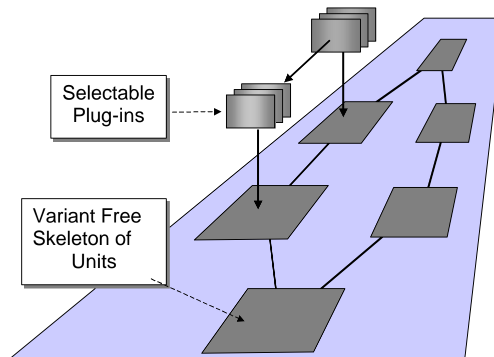

Figure 37. An architecture in the medical domain

5.5.5 Example 3: A Framework Architecture

Wijnstra [112] describes a software architecture in the medical domain that deploys component frameworks to manage diversity. Figure 37 provides a schematic overview of this architecture. It consists of a fixed structure of *units*, COM components (mostly executables) running on a Windows NT computer. Each unit is a component framework in which COM components can be plugged to implement the diversity of the product family. Plug-in components may talk to each other, but the initial contact is made through the main units.

In this architecture, the fixed structure of units[10] implements the common part of the family, while the plug-ins take care of the differences between the products. It is therefore an example of variation rather than composition (see the left hand side of Figure 31). Of course, this is true if the plug-ins are relatively small and specific for one product only. In real life, some plug-ins are rather large and shared between multiple members of the family. In fact, if one can create products by taking

[9] It is possible to run the desktop in a different process to prevent such crashes.

[10] Some units are optional.

arbitrary combinations of plug-ins, then this system exhibits signs of composition rather than variation.

5.6 Composition Further Explained

We shall now discuss composition as a technique for managing diversity.

5.6.1 What is Composition?

We speak of *composition* if two pieces of software that have been developed *without* direct knowledge of each other, can be combined to create a working product. The components may have been developed in some common context, or they may share interfaces defined elsewhere. The essential element is that the first piece of software can be used without the second, and the second piece without the first.

The underlying argument for using composition to handle diversity is the large number of combinations that can be made if components are not directly dependent on one another. Note that composition excludes the notion of plug-ins, since these are directly dependent on the underlying framework, and cannot be used in isolation.

A nice analogy may be the problem of describing a language, where an infinite number of sentences must be produced with a finite (and hopefully small) set of 'components'. Typically, such components are production rules in a grammar, where the production rules function both as a composition hierarchy as well as a set of constraints on the composition.

Garlan has illustrated how difficult composition can be in his well-known paper on Architectural Mismatches [33]. It appears that two components being combined must at least share some common principles to work together efficiently. We call this a lightweight architecture, but leave it to another paper to elaborate on how to create such an architecture.

Szyperski [103] defines components as 'having explicit context dependencies only', and being 'subject to composition by third parties'. In Koala [72], the former are called (explicit) requires interfaces, while the latter is called third-party binding. Tony Williams recognized the same two elements in an early paper leading to OLE [113], though an interface is called a 'base class' there, and the third party a 'creator'.

In the following subsections we provide examples of compositional approaches.

5.6.2 Example 1: The 'Pipes and Filters' Architectural Style

One of the oldest examples of composition to create a diversity of applications with only few components, is the pipes and filters architecture of Unix. With a

minimum of common agreements: character based I/O, a standard input and output stream, and a new line as record separator, with a small set of tools such as *grep*, *sed*, *awk*, *find* and *sort*, and with essentially one simple composition operator, the '|', one can program a large variety of tasks, including small databases, automatic mail handling and software build support.

Pipes and filters are considered an architectural style in [98], and indeed there are more examples with similar compositional value. Microsoft's DirectShow [58] is a streaming architecture that allows the creation of 'arbitrary' graphs of nodes that process audio and video signals. The nodes, called *codecs*[11], are implemented as COM servers, and may be obtained from different vendors. The graph can be constructed manually with a program that acts as a COM client. A simple type mechanism allows automatic creation of parts of such a graph, depending on types of input and output signals. Other companies have similar architectures, e.g. Philips with the Trimedia Streaming Software Architecture (TSSA) [106].

A third kind of pipes and filters architectures can be found in the domain of measurement and control. LabView [64] is an environment in which measurement, control and automation applications can be constructed as block diagrams and subsequently compiled into running systems. See also [16].

5.6.3 Example 2: Relation Partition Algebra

Software architectures are normally defined a priori, after which an implementation is created. But it is also possible to 'reverse engineer' an implementation to extract a software architecture from it, if the architecture has been lost. Even if the architecture has not been lost, it is still possible to extract the architecture from the implementation and verify it against the original architecture. Discrepancies mean that either the implementation is wrong, or the original architecture is 'imperfect' (to say the least).

Tools can be built to support this process, but a large variety of queries would result in a large range of tools. As it turns out, mathematical set and relation algebra can be used to model 'use' and 'part-of' relations in software effectively; queries can be expressed mostly in one-liners. A compositional tool set for this algebra is described in [79]; a similar technique is also deployed by [35].

5.6.4 Example 3: COM, OLE and ActiveX

Microsoft's COM [94] is a successful example of a component technology, but this does not automatically imply that it supports *composition* from the ground up. COM can be seen as an object-oriented programming technique, its three main functions, *CoCreateInstance*, *QueryInterface* and *AddRef/Release*, being

---

[11] For *co*ding and *dec*oding of streams of data.

respectively the new operator, the dynamic type cast, and the delete operator. COM's main contribution is that it supports object orientation[12] 'in the large': across language, process and even processor boundaries.

If a COM client uses a COM service, the default design pattern is that it creates an instance of that service. It refers to the service by CLSID, a globally unique identifier that is nevertheless specific for that service, thus creating a fixed dependency between the client and the service. In other words, clients and services are (usually) not freely composable. To circumvent problems thus arising, some solutions have been found, most noteworthy the *CoTreatAs*, the *Category ID* and the connectable interfaces. The latter provide a true solution, but are in practice only used for notifications.

Microsoft's OLE [17] *is* an example of an architecture that allows free composition, be it of document elements (texts, pictures, tables) rather than software components. An OLE *server* provides such elements, while an OLE *container* embeds them. An application may be an OLE server and container at the same time, allowing for arbitrarily nested document elements. An impressive machinery of interfaces is needed to accomplish this, as a consequence, there are not many different OLE containers around (though there are quite a few servers).

Microsoft ActiveX controls are software components that are intended for free composition. In general, they do not have matching interfaces, making it necessary to glue them together. The first language for this was Visual Basic; later it became possible to use other languages as well. An impressive market of third party ActiveX controls has emerged.

5.6.5 Example 4: GenVoca, Darwin and Koala

GenVoca [8] is one of the few approaches that supports explicit *requires* interfaces to be bound by third parties. GenVoca uses a functional notation to combine components and is equipped with powerful code generation techniques. It has been applied to e.g. graph processing algorithms, where different combinations of the same set of components can be made to create a variety of applications.

Darwin [52] is a research language for the specification of distributed software architectures. It supports components with typed *provides* and *requires* interfaces, and it allows compound components to instantiate and bind other components. Darwin has been applied to the control and monitoring of coal mining, and - as Koala - to control software for televisions (see below).

Koala [72] is an industrial architectural description language with a resource friendly component model used for the creation of embedded control software for consumer products. Koala supports both composition and variation. Koala

[12] Be it without implementation inheritance, a 'mishap cured in .NET'???

components can be instantiated and their (typed) interfaces can be bound by compound components as shown in Figure 38, thus forming a hierarchy of components. Variation is supported in the form of diversity interfaces, a special form of requires interfaces that resemble properties in ActiveX controls, and switches, a kind of pseudo dynamic binding. Various products are already on the market, created from different combinations of different subsets of a repository of reusable components. Koala supports adaptation in the form of glue modules as a fundamental technique for binding components.

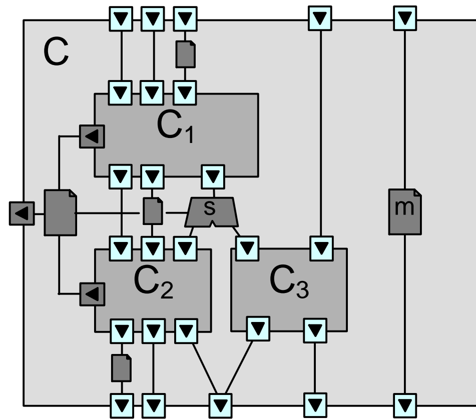

Figure 38. The Koala component model

5.6.6 Composability is Actually a Form of Variation!

We have discussed variation and composition. We presented them as fundamentally different, but it is possible to unify the concepts. In all examples that we have seen, components become 'freely composable' when context dependencies have been turned into explicit variation points. This may be in the form of an input stream or of connectable (requires) interfaces. Some approaches make every context dependency explicit (e.g. Koala), while others only make certain dependencies explicit (like in the pipes and filter architectures).

5.7 Concluding Remarks

Let us first recapitulate. Managing complexity and diversity while maintaining high quality and short lead-times have become important software development issues. Reuse, architecture and components are key solutions to achieve this. Originally, we envisaged reuse to happen 'automatically' once software components were available on the market. Later we discovered that sharing of software actually requires a pro-active, systematic approach: that of software product lines. The first product lines rely heavily on (limited) *variation* in an otherwise variant-free architecture. With product scope increasing, and software

development starting to cross-organizational borders, we believe that more powerful techniques are required to handle diversity, and discuss *composition* as one such a technique.

This does not mean that we're back to third party component markets. Instead a careful balance between architecture-driven top-down and component-driven bottom-up development is required, and we explain why and how. Also, composition does not replace variation, but rather they should be seen as two independent dimensions. We typify various points in the two-dimensional space with archetype examples, before explaining variation and composition individually in more detail. To make things as concrete as possible, we illustrate variation and composition with a rich set of well-known examples.

Within Philips, we have moved from variation to composition in our architectures for certain sets of consumer products. We find that we can handle diversity better, and cover a larger range of products even though development is decoupled in time and space across organizations. We find that few other companies use composition, which surprised us originally. We then discovered that the few companies that do use composition also cover a wide product scope, or cross-organizational boundaries, or exhibit both phenomena.

5.7.1 Acknowledgements

We would like to thank attendants of ARES workshops, the SPLC conference, and the Philips 'Composable Architectures' project for many fruitful discussions on this topic.