Chapter 6

Configuration Management

Published as: *Configuration Management in Component Based Product Populations*, Rob van Ommering, 10th International Workshop on Software Configuration Management, May 14-15, Toronto, 2001, Canada, also published in LNCS 2649, p16-23.

Abstract: The ever-increasing complexity and diversity of consumer products drives the creation of product *families* (products with many commonalties and few differences) and product *populations* (products with many commonalties but also with many differences). For the latter, we use an approach based on composition of software components, organized in packages. This influences our configuration management approach. We use traditional CM systems for version management and temporary variation, but we rely on our component technology for permanent variation. We also handle build support and distributed development in ways different from the rest of the CM community.

6.1 Introduction

Philips produces a large variety of consumer electronics products. Example products are televisions (TVs), video recorders (VCRs), set-top boxes (STB), and compact disk (CD), digital versatile disk (DVD) and hard disk (HD) players and recorders. The software in these products is becoming more and more complex, their size following Moore's law closely [13]. Additionally, the market demands an increasing diversity of products. We have started to build small software *product families* to cope with the diversity within a TV or VCR family. But we believe that our future lies in the creation of a *product population* that allows us to integrate arbitrary combinations of TV, VCR, STB, CD, DVD and HD functionality into a variety of products.

This paper describes the configuration management approach that we devised to realize such a product population. We first define the notion of product family and product population, then describe some of our technical concepts. We subsequently discuss five aspects of configuration management, and end with some concluding remarks.

6.2 Product Family and Population

We define a *product family* as a (small) set of products that have many commonalties and few differences. A software product family can typically be created with a single variant free architecture [88] with explicit variation points [41] to control the diversity. Such variation points can be defined in advance, and products can be instantiated by 'turning the knobs'. Variation points can range from simple parameters up to full plug-in components. An example of a product family is a family of television products, which may vary - among other things - in price, world region, signal standard and display technology.

We define a *product population* as a set of products with many commonalties but also with many differences [73]. Because of the significant differences, a single variant free architecture no longer suffices. Instead we deploy a composition approach, where products are created by making 'arbitrary' combinations of components. An example product population could contain TVs, VCRs, set-top boxes, CD, DVD and HD players and recorders, and combinations thereof. These products have elements in common, e.g. a front-end that decodes broadcasts into video and audio signals. These products are also very different with respect to for instance display device and storage facility.

It is true that any product population of which *all* members are known at initiation time, can in principle be implemented as a product family, i.e. with a single variant free architecture. In our case, two reasons prevent us from doing so. First of all, our set of products is dynamic - new ideas for innovative products are invented all the time. Secondly, different products are created within different sub organizations, and it is just infeasible - both for business as well for social reasons - to align all developments under a single software architecture.

6.3 Technical Concepts

In this section we briefly describe the software component model that we use, and show how components are organized into packages and into a component based architecture.

6.3.1 The Koala Component Model

We use software components to create product populations. Our component model, Koala, is specifically designed for products where resources (memory, time) are not abundant [70], [72]. Basic Koala components are implemented in C, and can be combined into compound components, which in turn can be combined into (more) compound components, until ultimately a product is constructed. The Koala model was inspired by Darwin [52].

Koala components have explicit provides *and* requires interfaces. Provides interfaces allow the environment of the component to use functionality

implemented within the component; requires interfaces allow the component to use functionality implemented in the environment. All requires interfaces of a component must be bound explicitly when instantiating a component. We find that making all requires interfaces both explicit and third-party bindable, greatly enhances the reusability of such components.

Provides and requires interfaces of components are actually instances of reusable *interface definitions*. We treat such interface definitions as first class citizens, and define them independently from components. Such interface definitions are also common in Java, and correspond with abstract base classes in object-oriented languages.

Diversity interfaces, a special kind of requires interfaces, play an important role in Koala. Such interfaces contain sets of parameters that can be used to fine-tune the component into a configuration. We believe that components can only be made reusable if they are heavily parameterized, otherwise they either become too product specific or they become too generic and thus not easily or efficiently deployable.

Note that the Koala terminology slightly differs from that used by others. A Koala component *definition*, or more precisely a component *type* definition, corresponds closely with the notion of *class* in object-oriented languages. A Koala component *instantiation* corresponds with an *object*. What Szyperski calls a *component*, a binary and independently deployable unit of code [103], resembles mostly a Koala package as described in the next section, i.e. a set of classes and interfaces. It is true that we distribute source code for resource constrained reasons, but users of a package are not allowed to change that source code, only compile it, which inherently is what Szyperski meant with 'binary'.

6.3.2 Packages

We organize component and interface definitions into *packages*, to structure large-scale software development [75]. A package consists of public and private component and interface definitions. Public definitions can be used by other packages - private definitions are for use in the package itself only. This allows us to manage change and evolution without overly burdening the implementers and users of component and interface definitions.

Koala packages closely resemble Java packages, where component definitions in Koala resemble classes in Java, and interface definitions in Koala resembling interfaces in Java. Unlike in Java, the Koala notion of package is *closed*; users of a package cannot add new elements to the package.

6.3.3 The Architecture

With Koala we have created a component-based product population that consists of over a dozen packages, each package implementing one specific sub domain of

functionality [74]. Products can be created by (1) selecting a set of packages appropriate for the product to be created, (2) instantiating the relevant public component definitions of those packages, and (3) binding the diversity and (4) the other requires interfaces of those components. Product variation is managed by making choices in steps (1) - (4).

6.4 Configuration Management

We now come to the configuration management aspects of our product population development. We distinguish five issues that are traditionally the domain of configuration management systems, and provide our way of dealing with these issues. These issues are:

- Version management;
- Temporary variation;
- Permanent variation;
- Build support;
- Distributed development.

The issues are discussed in subsequent subsections.

6.4.1 Version Management

In the development of each package, we use a conventional configuration management system to maintain a version history of *all* the programming assets in that package, such as C header and source files, Koala component and interface definitions, Word component and interface data sheets, et cetera. Each package development team deploys its own configuration management system - see also the section that describes our distributed development approach.

A package team issues formal *releases* of a package, where each release is tagged with a package version identification (e.g. 1.1b). The version identification consists of a major number, a minor number, and a 'patch letter'. The major and minor version numbers indicate regular releases in the evolution of the package - the 'patch letter' indicates bug-fix releases for particular users of the package.

Each formal release of a package must be internally consistent. This means that in principle *all* components should compile and build, and run without errors. In practice, not every release is tested completely, but it *is* tested sufficiently for those products that rely on this specific version of this package.

Customers of a package only see the formal releases of the package. They will download the release from the intranet (usually as ZIP file), and insert it in their local configuration management system. They will only maintain the formal

release history of the package, and will not see all of the detailed versions of the files in the package.

Each release of a package must be backward compatible with the previous release of the package. This is our *golden rule*, and it allows us to simplify our version management - only the last release of each package is relevant. Again in practice, it is sometimes difficult to maintain full backward compatibility, so sometimes we apply our *silver rule*, which states that all existing and relevant clients of the package should build with the new release of the package.

6.4.2 Temporary Variation

For temporary variation we use the facilities of traditional configuration management systems. We recognize two kinds of temporary variation.

Temporary variation *in the small* arises if one developer wants to fix a bug *while* another developer adds a feature. By having developer specific branches in the CM system, developers do not see changes made by other developers *until* integration time. This allows them to concentrate on their own change before integrating changes made by others into their system.

Temporary variation *in the large* arises just before the release of a product utilizing the package. Note that packages are used by multiple products - this is what reuse is all about. We cannot run the risk of introducing bugs into a product coming out soon when adding features to a package for a product to be created somewhere in the future. This is why we create a *branch* in the package version history for a specific product just before it is being released. Bug fix releases for such a product are tagged with 'patch letters', e.g. 1.2a, 1.2b et cetera. As a rule, no features may be added in such a branch, only bugs may be fixed.

Although the branch may live for a number of weeks or even months - it may take that long to fully test a product - we still call this a temporary branch since the branch should be closed when the product has been released. All subsequent products should be derived from the main branch of the package. There is also *permanent variation* in our product population, but we handle that completely *outside* of the scope of a configuration management system, as described in the next section.

6.4.3 Permanent Variation

For permanent variation, i.e. the ability to create a variety of products, we do not deploy a traditional configuration management system, but rely on our component technology. We have a number of arguments for doing so:

- It makes variation explicit in our architecture, instead of hiding it in the CM system

- It allows us to make a late choice between compile time diversity and run-time diversity, whereas a CM system typically only provides compile time diversity.
- It allows us to exercise diversity outside the context of our CM system (see also section 6.4.4).

The second item may require further clarification. Koala diversity interfaces consist of parameters that can be assigned values (expressions) in the Koala component description language. The Koala compiler can evaluate such expressions and translate the parameters into compile-time diversity (#ifdefs) if their value is known at compile time, or run-time diversity (if statements) otherwise. This allows us to have a *single* notion of diversity in our architecture. We can then still decide at a late time whether to generate a ROM for a *single* product (with only the code for that product), or a ROM for a set of products (with if statements to choose between the products). Note that always generating a ROM for a single product complicates logistics, but on the other hand always generating a ROM for multiple products increases the ROM size and hence the bill of material.

Put differently, we expect that once (binary) components are commonplace, the whole configuration issue will be handled at run-time, and not by a traditional CM system.

With respect to the first item mentioned above, we have four ways of handling diversity [75]:

- Create a single component that determines by itself in which environment it operates (using *optional* interfaces in Koala, modeled after *QueryInterface* in COM).
- Create a single component with diversity interfaces, so that the *user* of the component can fine-tune the component into a configuration;
- Create two different (public) components in the same package, so that the *user* of the package can select the right component;
- Create two different packages for different implementations of the same sub domain functionality, so that *users* may select the right package (before selecting the right public component, and then providing values to diversity parameters of that component).

It depends on individual circumstances which way is used in which part of the architecture.

6.4.4 Build Support

The fourth issue concerns build support, a task normally also supported by traditional CM systems. We deliberately uncoupled the choice of build environment from the choice of CM system, for a number of reasons.

First of all, we cannot (yet) *standardize* on one CM tool in our company. And even if different groups have the same CM system, they may still utilize incompatible versions of that system. So we leave the choice of a CM system open to each development group (with a preferred choice of course, to reduce investments). This implies that we must separate the build support from the CM system.

Secondly, we do not *want* to integrate build support with CM functionality, because we want to buy the best solution for each of these problems, and an integrated solution rarely combines the best techniques. So our build support currently utilizes our Koala compiler, off-the-shelf C compilers and a makefile running on multiple versions of make. We deploy Microsoft's Developer Studio as our IDE.

Thirdly, we want to be able to compile and link products while *outside* the context of a CM system, e.g. when in a plane or at home, or in a research lab. Although not a compelling argument, we find this approach in practice to be very satisfying, because it allows us to do quick experiments with the code at any place and at any time.

6.4.5 Distributed Development

The fifth and our final issue with respect to configuration management concerns distributed development. Recently, many commercial CM systems provide support to distribute the CM databases over the world. We are not in favor of doing that for the following reasons.

First of all, it forces us to standardize again on *one* CM system for the entire company, and we have shown the disadvantages of this in the previous section.

Secondly, we want to be able to utilize our software *outside* the context of the CM system, and relying on a distributed CM system does not help here.

Thirdly, we think this approach doesn't scale up. We envisage a 'small company approach' for the development of packages in our company. You can 'buy' a package, and download a particular release of that package, *without* having to be integrated with the 'vendor' of that package in a single distributed CM system. Imagine that in order to use Microsoft Windows, you would have to be connected to their CM system!

This concludes the five issues that we wanted to tackle in this paper.

6.5 Concluding Remarks

We have explained our component-based approach towards the creation of product populations. We briefly sketched our Koala component model and the architecture that we created with it. We then listed five configuration management issues, some of which we may have tackled in different ways than is conventional. The issues are:

- Version management of files and packages;
- Temporary variation for bug fixes, feature enhancements and safeguarding products;
- Permanent variation as explicit issue in the architecture;
- Build support separated from configuration management;
- Distributed development using a 'small company model' rather than a distributed CM system.

The approach sketched in this paper is currently being deployed by well over a hundred developers within Philips, to create the software for a wide range of television products.