

## Variant Management

### Mark Dalgarno

Software Acumen Limited  
St. John's Innovation Centre,  
Cowley Road,  
Cambridge,  
CB4 0WS  
England  
Telephone: +44 (0)1223 323326  
mark@software-acumen.com

### Dr. Danilo Beuche

pure-systems GmbH  
Agnetenstr. 14  
39106 Magdeburg  
Germany  
Telephone: +49 391 544 569 0  
Fax: +49 391 544 569 90  
danilo.beuche@pure-systems.com

## Abstract

Related products frequently share much of the same software, with only a few differences realizing product-specific functionality. However, much of the challenge of developing related products comes from managing these differences.

When faced with this challenge, organisations often turn to their Configuration Management (CM) system to manage the variability between products. We will illustrate this using case studies, and then discuss the reasons why this approach is rarely wholly successful.

We will then consider another case study, of an organisation that recognized that use of CM alone was insufficient to manage variability. This organisation has adopted a Variant Management approach, which addresses the problems of a pure CM approach, by enabling the development of a group of related products as a whole, rather than as individual, independent projects. CM still plays an essential role in this organisation but this role has changed with the introduction of Variant Management.

**Keywords:** variant management, configuration management, software product lines, software families, software reuse, software process improvement

## 1. Introduction

One increasing trend in software development is the need to develop multiple, similar software products (a *Software Product Line*) instead of just a single product. There are several reasons for this. Products that are being developed for the international market must be adapted for different legal or cultural environments, as well as for different languages, and so must provide adapted user interfaces. In the embedded world differing customer needs and differences in device types constrain or otherwise dictate the need for different software variants for each device.

Because of cost and time constraints it is not possible for software developers to develop a new variant from scratch for each new product and so software reuse must be increased. These types of problems commonly occur in portal or embedded applications, e.g. vehicle control applications [Steger04] but are also seen in desktop applications.

In these examples the different product variants frequently share much of the same software, with only a few differences realizing product-specific functionality. However, much of the challenge of developing product variants comes from managing these differences.

In Section 2 of this paper we describe how organisations faced with this challenge often turn to their Configuration Management (CM) system for help. We describe a variety of commonly-found approaches and discuss the pros and a cons of each approach. We then conclude this section by noting a number of general problems that can occur when attempting to manage product variants **solely** through CM.

In Section 3 we describe *Variant Management* an emerging approach for managing the development of product variants that addresses the problems of a pure CM approach by enabling the development of a group of related product variants **as a whole**, rather than as individual, independent projects.

In Section 4 we illustrate adoption of Variant Management with a case study from an organisation that recognised that CM alone was insufficient to manage differences between product variants.

## **2. Managing variants with CM**

As noted above, when faced with the challenge of managing multiple product variants, organisations often turn to their CM system to manage the differences between product variants e.g. [Jackson00] and [HKM06].

We will describe three common approaches below and discuss the pros and cons of each approach. To support this description we introduce some terminology:

*Configuration Item:* A discrete software element<sup>1</sup> whose development is to be controlled.

*Configuration:* - A coherent set of configuration items. A *Product Configuration* is a releasable configuration.

*Version:* Versions are instances of **a single** configuration item that have undergone some change (or revision). Only one version of a configuration item is typically present in any given configuration.

*Variant:* Variants are instances of **different** configuration items that are similar-but-different; they typically represent alternative functional

---

<sup>1</sup> We use the term software element here for convenience although the approaches we describe are not specific to software.

capabilities. Only one variant from a set of similar-but-different variants is typically present in any given configuration.

*Product Variants:* Product Variants are alternative product configurations, for example for different customers, different markets or with alternative variants included.

We will now describe three common approaches for managing variants using CM – *Clone-and-Own*, *Independent Component Teams* and *Platform Versions*.

## **2.1. Clone-and-Own**

Perhaps the simplest approach to developing product variants is *Clone-and-Own*. In this approach new product variants are created by making an exact copy of all the configuration items of an existing product variant on a new codeline (or branch). The new product variant is developed by making modifications that implement the differences between the old and the new product variants [Beuche07].

The advantage of Clone-and-Own is that once a new codeline has been created for a product variant it can be developed completely independently of other work. However, the downside is that this approach fails to recognise that variants typically share much functionality. For example, if shared functionality has to be developed after a new codeline has been created, e.g. to fix a bug or implement an enhancement, then it has to be copied from one codeline to another. In the worst case this does not happen, the two variants diverge further and the maintenance burden on the organisation increases.

## **2.2. Independent Component Teams**

A second approach that addresses some of the problems of Clone-and-Own is to subdivide development of software into relatively independent component development teams [Jackson00], [Beuche07]. Each component is developed on its own codeline and released periodically. New variants are created by integrating some subset of the components and possibly adding some product-specific code.

In some cases this picture is further complicated by development of component variants as branches of the main component development codeline [HKM06] but we will not explore this complexity here.

Like Clone-and-Own this approach allows teams to progress independently of each other and can work well when component releases are well coordinated or when components have relatively simple dependencies, for example if components are widely used in many projects in a library-like style.

However, integration problems can arise when components have complex dependencies or when they require different build contexts [Jackson00]. Imagine a situation where *Component A* version 1.0 works with *Component B* version 1.0 or *Component B* version 2.0, but requires *Component C* version 3.1 if *Component B* version 2.0 is used. With complex dependencies it can be hard to establish what the valid configurations of

components are and pervasive changes can be hard to manage [ibid.]. Furthermore, the process of releasing components can become burdensome and so can act as a drag on developers, thereby negating the benefits of allowing teams to work independently [ibid.].

### 2.3. Platform versions

A third approach brings component development into a single codeline (a mainline) [JB03]. When a defined set of features has been completed a release branch is made from the mainline, tested, and after any bugs have been fixed (on the release branch) is shipped to the customer as a new **version** of the platform [ibid.].

However, as noted in [JB03], customers may be reluctant to take new platform versions due to the effort required to accept the new platform or integrate it with their own software. When variants are treated as versions this arises in part because later platform versions will typically include changes that are not required by that customer and these can have a destabilising effect.

The upshot of this is that customers will ask for new versions of their existing platform version to be developed in preference to taking the current latest platform version. There will also be pressure to migrate **some** later changes to the platform back into their existing platform version (known as *backporting*) [ibid.]. Figure 1 from [JB03] nicely illustrates these issues.

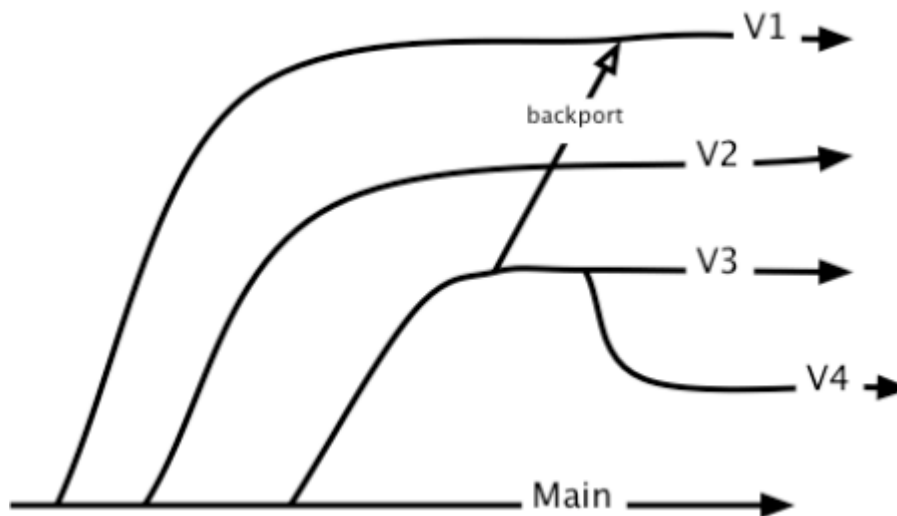


Figure 1: Problems with Platform Versions (From [JB03])

Bowing to customer pressure extends the life of earlier platform versions and so increases the cost of maintaining those versions. Backporting is also an expensive activity. Developers have to implement the same feature each time they backport. Aside from wasted time and effort this is not necessarily problem-free since the feature may have to be implemented in a (subtly) different way each time it is backported.

One final point to make is that in the real world hybrids of these various approaches can be found. For example, Clone-and-Own can be done partially with only some configuration items being copied, branched and modified while others are developed in shared code lines. This hybrid has the mixed problems of Clone-and-Own and Independent Component Teams. The number of unshared but should-be-shared elements can be minimized, but at the expense of complex dependencies when making changes to a shared item.

#### ***2.4. Could you have a variant management problem?***

From the descriptions above we can see that the following issues can **indicate** that you **may** have a variant management problem rather than a version management problem:

- You have an increasing number of independently-developed CM branches.
- You have an increasing amount of code in independently-developed CM branches.
- You are fixing the same bug or developing the same enhancement more than once.
- Bugs and enhancements developed on one branch are not making their way to other branches.
- The dependencies between software components are unclear. You cannot explain why some components can only be integrated in certain ways.
- You have an increasing number of failed integrations. You cannot explain why some integrations fail.
- You have an increasing maintenance cost.
- Your customers are asking for new versions of your earlier releases in preference to taking a later release.
- Your customers are asking you to backport functionality from later releases into earlier releases.

In our general experience several other indicators may also be present:

- Your CM and / or build processes are becoming more complex or harder to maintain. Product builds may only be possible with expert knowledge.
- You already have some product variants but there is no cohesive platform from which they are developed and / or they are diverging in unplanned ways.
- You already have some product variants but there is an increasing demand for product differentiation – either more product variants or more differences between the variants.

### 3. Variant Management

Variant Management is an emerging approach that addresses the problems of a pure CM approach by enabling the development of a group of related products as a whole, rather than as individual, independent projects. CM still plays an essential role in an organisation using Variant Management but this role changes with the introduction of Variant Management.

Variant Management works by supporting a logical separation between the development of core, reusable software assets (the platform) and application (product) variants. During application development, platform software is selected and configured to meet the specific needs of the application.

The common and variable features of product variants are described in the Problem Space. This reflects the desired range of applications (product variants) in the Product Line (the “domain”) and their inter-dependencies. So, when producing a product variant, the application developer uses the Problem Space definition to describe the desired combination of problem features to implement the product variant.

An associated Solution Space describes the constituent assets of the Product Line (the “platform”) and its relation to the Problem Space, i.e. rules for how configuration items in the platform are selected when certain values in the problem space are selected as part of a product variant. The four-part division resulting from the combination of the Problem Space and Solution Space with domain and application engineering is shown in Figure 2.

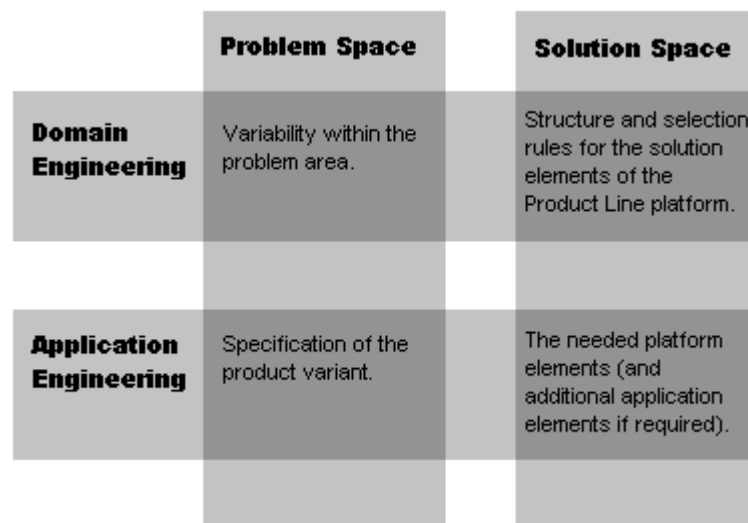


Figure 2: Overview of SPLE activities

Several different options are available for modelling the information in these four quadrants. The Problem Space can be described with Feature Models, or with a Domain Specific Language (DSL). There are also a number of different options for modelling

the Solution Space for example component libraries, DSL compilers, generative programs and also configuration files [Czarnecki00].

When such models exist tool support can be provided to support the definition of individual product variants by specifying values for any variable feature. Such tools can guarantee the correctness and completeness of an individual variant specification and so reduce the time required to specify a product variant. This also means that less specialist knowledge is needed by individuals to create such a specification as the knowledge is embodied in the model.

Configuration knowledge can also be captured by associating configuration items in the Solution Space with features in the Problem Space. For example, *the 'Temperature Component' is only present when the 'Measure Temperature' feature has been specified for the product variant*. With tool support, techniques such as assembling shared (common) and unshared (variable) file **fragments** can also be used safely and reliably. This avoids the need to create separate variants to handle differences between different product variants, or can at least minimise the amount of code in each variant and so reduces the amount of branching in the code base and / or the amount of code that has been branched.

#### **4. Adopting Variant Management**

[Beuche07] describes the case of an organisation that faced some of the above issues and ultimately addressed them by adopting a Variant Management approach. At the beginning of the time period covered by the study the organisation produced a number of product variants and had recognised that managed reuse could help improve productivity.

Its first attempt was a partial success. A software architecture was developed and this was reused in different product variants. However, individual product variants were developed using a *Clone-and-Own* approach as described above, with the consequent problems.

A few years later, in the face of increasing demand for more product variants and for greater differentiation between product variants – indicators that they had a variant management problem – the organisation instituted a platform development project to try and create a common software base from which to derive all product variants.

One initial concept for this was the *Independent Component Teams* approach as described above. However there were doubts within the team about the technical feasibility of this approach in their organisation.

After further investigation of the literature and participation in a specialist Software Product Lines conference the team chose to investigate whether a Variant Management approach based on a common platform and an associated feature model could address their requirements.

Their investigation began by working with two separate codelines for two product

### 3rd British Computer Society Configuration Management Specialist Group Conference

variants to identify common and variable code. They used the GNU *diff* application to generate a merge of the two codelines and identified 1003 files which were identical in both variants, 281 files which contained some common and some variable code (implemented as C *#ifdef* directives), and a smaller number of files that were unique to either product or were so different it didn't make sense to merge them. Once this merge had occurred the team began to refactor the shared code so that the *#ifdef* directives were redefined in terms of features (from the associated feature model) rather than product names.

Some time into this refactoring the team recognised that a quick platform release could bring benefits. Some specific subsystems were identified as the basis for the initial platform release based on a strong desire by the platform team to control variation in those subsystems. As it turned out this initial platform represented around 60% of the code base.

Using this approach, new features are developed on their own branches and merged into variant branches or the platform as desired. In order to increase the scope of the platform a platform group was made responsible for coordinating migration of features from product variant projects to the platform. As well as making the feature available for everyone who needs it this will ultimately lead to fewer branches and less branched code. The result is that there are fewer long-living branches and improved reuse. A side-effect is that product variant projects also have the freedom to add new features they are developing to their build if required – for example if a customer demands a quick feature integration release.

## 5. Summary

Organisations frequently turn to their CM system when faced with producing product variants. We have illustrated three different patterns of CM use – *Clone-and-Own*, *Independent Component Teams* and *Platform Versions* – that attempt to manage the production of product variants and discussed the pros and cons of each approach.

We then described Variant Management, an emerging approach that addresses the problems of a pure CM approach by enabling the development of a group of related products as a whole, rather than as individual, independent projects.

Variant Management allows robust integration of diverse configuration items to create product variants. By supporting systematic merging of file fragments and techniques such as code generation Variant Management also contributes to reduced branching and reduced code size in branches and so maximises software reuse and therefore addresses the problems of a pure CM approach while still retaining its strengths.

## 6. References

[Beuche07] D. Beuche, Unpublished experience report, 2007

[Czarnecki00] K. Czarnecki, U.W. Eisenecker, *Generative Programming: Methods*,

**3rd British Computer Society Configuration Management Specialist Group Conference**

*Tools, and Applications*, Addison-Wesley, 2000

[HKM06] W.A. Hetrick, C.W. Krueger, J.G. Moore, *Incremental Return on Incremental Investment: Engenio's Transition to Software Product Line Practice*, OOPSLA '06 October 22-26, 2006

[Jackson00] P. Jackson, *A year with Perforce*,  
<<http://www.perforce.com/perforce/us/2000/jackson/jackson.html>> accessed February 2007

[JB03] P. Jackson, R. Brooksby, *Changing how you Change*,  
<<http://www.ravenbrook.com/doc/2003/03/06/changing-how-you-change/>> accessed February 2007

[Steger04] M. Steger et al., *Introducing PLA RK Bosch Gasoline of System: Experiences and Practices*, in: Proc. of the Software Product Line Conf. 2004, S. 34-50