# Handling Variability

Version 2.0, December 11, 2009

Markus Völter, Independent/itemis
(voelter@acm.org)



## Introduction

Traditionally, in software engineering, development happens for single products. This is a very inefficient approach in cases where groups of products are related. Software product line engineering [1] is about systematically developing families of related products together, as a product line. The products within a product line usually have many things in common, but also significant differences. Managing and implementing these differences can become complex because in realistic product lines, variability abounds, and it is often a cross-cutting concern. Hence, to exploit the benefits of product line engineering, it is important to systematically manage the variability between the products.

Variability denotes differences between related products in a product line. Typically one talks about variation points, where, to define a product, you need to bind each variation point. There are different ways to bind a variation point: setting a value, selecting an option or implementing a program fragment or model (we'll talk about this in the patterns below). For each variation point, you'll also have to define the binding time: at design time, load time, runtime, etc.

This paper is a collection of patterns for handling variability in software systems. It contains patterns for managing variability, introduces different kinds of variability, and illustrates realization of variability in implementation artifacts such as models or source code. The patterns are intended as a contribution to a more comprehensive pattern language on product line engineering.

The paper is intended to be read by architects who want to get a better grasp on managing and implementing variability. The paper does not address requirements and product management. I assume the requirements that drive the variability are known.

# Structure of the Paper

The paper is structured into three sections, Managing Variability, Classes of Variability and Implementing Variability.

*Managing Variability* provides two approaches on how to reduce the overall complexity that results from variability. One pattern, SEPARATE DESCRIPTION OF VARIABILITY recommends the separation of the logical description of variability from its implementation. The other one, MODEL-BASED IMPLEMENTATION describes how and why to use domain-specific models to capture variability.

The second chapter, *Classes of Variability*, contains two patterns, CONFIGURATION and CONSTRUCTION. The level of expressiveness of these two approaches is fundamentally different, and you have to make a conscious decision for one of these when thinking about how to describe variability.

Chapter three, *Implementation Strategies*, deals with lower-level mechanisms for representing variability in implementation artifacts. It consists of three patterns: REMOVAL, where you conditionally take something away from a whole, INJECTION where you conditionally add something to a minimal core, as well as PARAMETERIZATION where you define a variant by providing values for a predefined set of parameters.

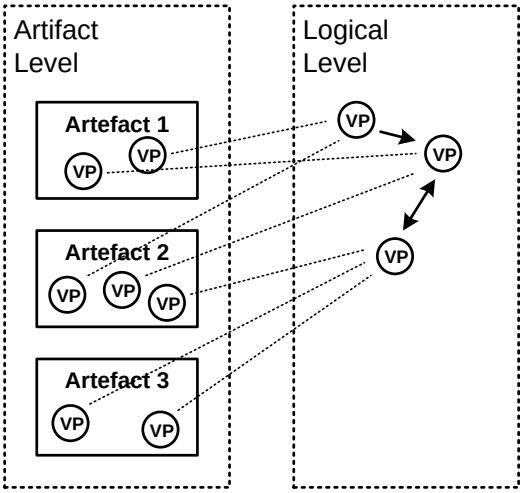
# Representing Variability

In large product lines with many products and many differences between the products, the variability inherent to implementation artifacts can easily overwhelm developers. The overhead for representing, organizing or managing the variability can become so complex that the potential benefits of product line engineering cannot be realized

**How can you represent the complexity introduced by variability in implementation artifacts?**

## Separate Description of Variability

**Make sure that the logical description of variability is separate (external) from its realization in the implementation artifacts. The logical description describes the variation points, the variants, as well as constraints between these variants. The realization of the variability in the implementation artifacts is tied to the logical variability description.**




- Artifact Level represents realization artifacts such as models, code or documentation
- The Logical Level is the external description of variation points and the conceptual constraints among them.
- VP's are variation points
- One or more VPs in the implementation level are associated with variation points in the logical level (n:1, n:m)

As a consequence of the cross-cutting nature of variability in many of today's systems, the implementation of variability is scattered over many implementation artifacts. However, in many cases several variation points need to be configured in a consistent, mutually dependend way for the resulting product to work. If each has to be configured separately, the overall complexity grows quickly. By identifying logical variation points, and then tying the (potentially many) implementation variation points to these logical variation points, related implementation variations can be tied together and managed as one. With reasonable tool support, you can also select a logical variation point and navigate to all the implementation variation points, providing a level of traceability. When customizing the artifact level based on a configuration of the logical level, the mapping should be automated, but doesn't have to be.

In most cases, the logical variability is also much more closely aligned with the problem domain. The variability in the artifacts corresponds to the solution domain. Consequently, meta data (why does the variability exist, which stakeholders care about it, etc.) is associated with the logical level. The logical level is typically visible to the non-developer stakeholders.

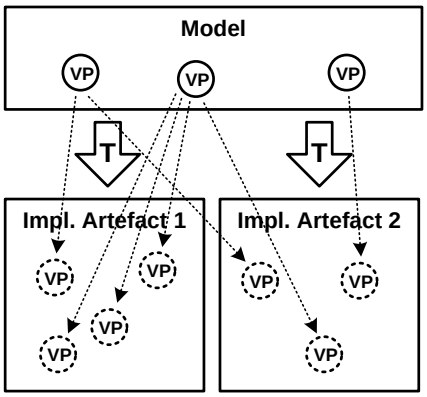
≈≈≈

One way of separating logical variations from implementation variations is using feature models. A feature model [2] describes features and their dependencies in a hierarchical fashion. Implementation artifacts or artifact processors can refer to those features and construct the product variant accordingly.

An alternative approach is OVM, or orthogonal variability models [3]. In contrast to feature models, they are not hierarchical. Technically, they don't describe features and their relationships but rather variation points. The two representations are semantically equivalent.

# Model-Based Implementation

**Describe the implementation of the system with high-level constructs, such as models based on domain specific languages, and a subsequent transformation, interpretation or code generation step. Because of the closer alignment with the actual problem domain, variability is much more localized, and the number of variation points is significantly reduced in models compared to code.**




- A model describes domain abstractions in a formal and concise way
- Transformations map that model to (typically more than one) implementation artifact
- Variability is expressed with fewer VPs in the models compared to implementation artifacts

**Model-Driven Development**

In Model-Driven Development, we develop domain-specific languages that are very closely aligned with the domain at hand. Consequently, when using such a DSL (Domain-Specific-Language) to describe a system in that domain, the resulting models/programs become very concise. There's much less repetition and low-level detail in the description.

In a subsequent step, code generation (and sometimes interpretation) is used to map the models to implementation code. The knowledge about how to "expand" the knowledge in the models to implementation code is encoded into the code generator.

If you can describe something with a smaller amount of "stuff" (i.e. code, configuration files, etc) on a more abstract domain specific level, and then use the transformation or generator to expand all the details, you can simply implement the variation on the more abstract level in one place. The trade-off is, of course, that you have to define this high-level domain specific language, including a way to define variants of programs written in that language. You also need to define the transformation down to the actual implementation artifacts.

The relationship of this pattern to SEPARATE DESCRIPTION OF VARIABILITY is interesting. As the name suggests, the models mentioned in this pattern play the role of the implementation/artifact level in SEPARATE DESCRIPTION OF VARIABILITY. The logical description "customizes" the models which are then further mapped down to code. In some cases the models in this pattern play the role of the logical level and are not further customized by an additional logical level.

≈≈≈

Consider the case where the attributes of an address entity need to be variable. For example, in the US version of the system the address needs to have a state attribute. In European countries this is not necessary. The state attribute needs to be taken into account in the UI, the data structures, the database table structure, the SQL code to persist the data, and maybe in several other places. Instead of implementing the variability in each of these places, you can simply put one variation into a model that describes the data structure, and then use

code generation to derive the UI, the data structure, the database table creation statements, as well as the SQL code from that model.

Another, similar example is the implementation of state-based behaviour. If it is implemented directly with a programming language, you have to use either the State pattern, a big switch/case statement, a number of arrays pointing into each other, or state tables, together with a number of constants representing the states, events and transitions. If the state-based behaviour should be variable, implementing this variability on the level of the implementation is very tedious and error prone. An alternative approach is to directly describe the behaviour as a state machine using a suitable language, together with an interpreter written in the target language. Making some of the states, events or transitions variable requires only *one* change (for each variability) in the model and no changes to the interpreter, reducing the overall complexity significantly.

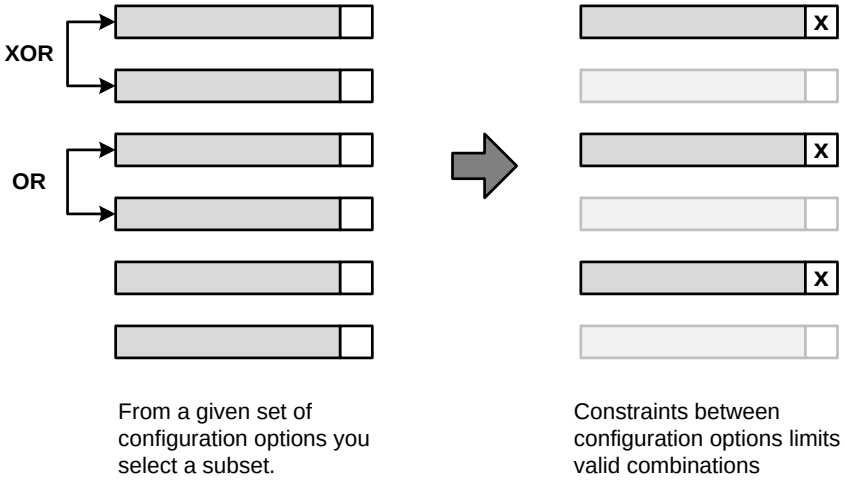
# Classes of Variability

Regarding the definition of variation points and the mechanisms to define the variants, there are several alternatives with different levels of expressiveness.

**How can the different alternatives be grouped according to their expressiveness?**

## Configuration

**A variation point allows the selection from several alternatives. Each alternative is either in the system or not. Constraints between the alternatives limit the valid combinations.**




The biggest advantage of configuration is its simplicity. People don't have to learn complex formalisms for defining a variant, they simply select from a predefined set of alternatives. Invalid selections are avoided by

restricting the valid combinations. To achieve this, constraints (*requires, prohibits, recommends, discourages*) are defined between the configuration options. Of course there are limits to what you can do with configuration only. For example, cardinalities, instantiation or relationships cannot be expressed very well. This can be seen as an advantage (makes the configuration process simple) and as a liability (the degrees of freedom are limited).

If you want end-users to configure your product, you should try to go as far as possible with configuration only.

≈≈≈

In the simplest case, configuration can be achieved by simply setting flags in a configuration file.

In C compilers, the ability to define symbols which are then evaluated by *ifdefs* is another way of configuration. Another alternative is using the Bridge or Strategy patterns. These support "plugging in" different implementations at a specific variation point. In contrast to preprocessors, they are bound at runtime using polymorphism in object oriented languages.

**The C Preprocesseor**

The C and C++ language family includes a preprocessor that can process the source code before it is submitted to the actual compiler.

One of the features of the preprocessor is to conditionally remove a region of code. To do that, you have to use #ifdef:

```
#ifdef aSymbol
  // here is some code
#endif
```

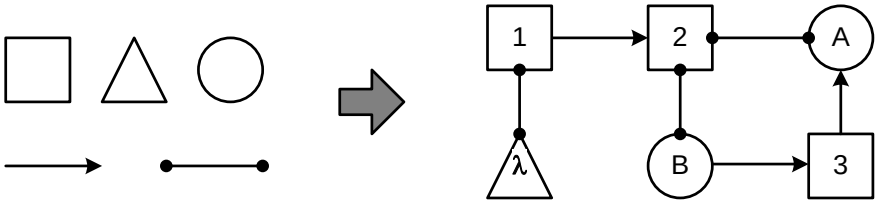
The code between the #ifdef and the #endif is removed (and hence, not compiled) if *aSymbol* is not defined. A symbol can be viewed as a boolean variable, and defining a symbol means to set it to *true*.

A more powerful formalism for configuration variability is feature models [2]. Feature models are hierarchical collections of flags (features) that can be selected or not. There are several default constraints between such features: mandatory (the feature must be included), optional (the feature might be included), alternative (exactly one of the set of features has to be included) and or (one or more from a collection of features has to be included). Powerful tools exist to manage even large sets of features and their relationships.

Most wizards are also a kind of configuration. You are guided through a number of selections and parameter specification. What you have selected in steps *1* through *n* possibly determines the options you can select from in options *n+1* through *k,* a form of constraints. From the resulting overall configuration some kind of artifact is generated or some functionality is executed.

## Construction

**A language is provided to define the variant. The definition of the variant is a sentence in this language.**




You define a language that can be used to define a basically an unlimited number of variants

You then define a sentence in that language that describes a particular variant

Construction is much more powerful than configuration, since it provides an unlimited variant space. The language defines a grammar (or a meta-model) and all valid instances are valid variants. Picture this in the illustration above: you can always add one more box and line. Depending on the language definition, construction can also be much more complicated to use than configuration, because of the unlimited variant space. However, it can be used to express relationships, instantiation and cardinalities.

≈≈≈

The most well-known example for construction is simply programming languages. Frameworks define hooks into which the developer can plug in code, as long as it conforms to a certain interfaces or other framework imposed constraints. Essentially, the variability is unlimited.

Whenever domain specific languages [4] are used to configure a product, then this is also construction. The variability is more limited, i.e. domain specific, but almost all DSL grammars allow for unlimited variability.

The composition of a system from components that are then hooked up in order to communicate is also a form of construction. This hooking up can, for example, happen through a dependency injection framework or through any other means of configuration file.

## Combinations

Of course, configuration and construction can be used in conjunction.

- A complex system can be subdivided into several subsystems, where possibly one set of subsystems is configured by a configuration and another set of subsystems will be configured by construction.
- Configuration can be superimposed onto construction, where a constructively created variant is customized by configuration. This can be achieved using Removal or Injection, as explained below.
- It is also feasible to use construction to provide details to configuration. Many configuration options have parameters (see Parameterization below). The type of such a parameter can be a

construction language. Every instance of a construction language would be a valid value for the parameter.

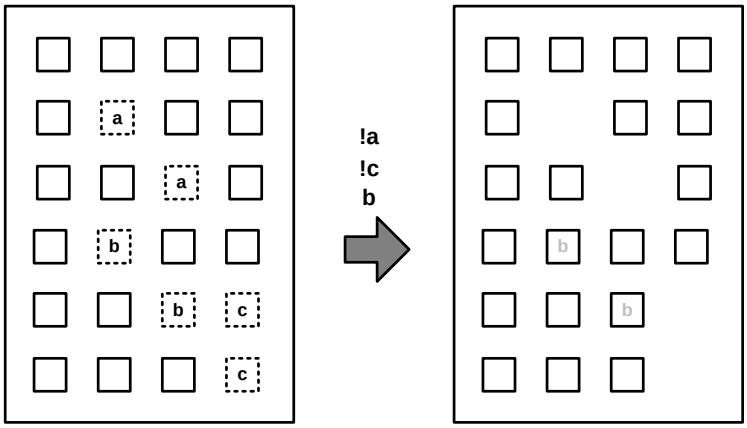
# Implementation Strategies

Now that we have defined the various classes of expressiveness we can look at the actual implementation of variability in implementation artifacts.

**How can variability be implemented in implementation artefacts?**

## Removal (aka negative variability)

**Remove parts of a comprehensive whole. This implies marking up the various optional parts of the comprehensive whole with conditions that determine when to remove the part.**




An artifact contains parts (the rectangles) annotated with features (a, b, c) with which they are associated

A variant removes those those parts whose features are not selected in the configuration

The biggest advantage of this approach is its apparent simplicity. However, the comprehensive whole has to contain the parts for all variants (maybe even parts for combinations of variants), making it potentially large and complex. Also, depending on the tool support, the comprehensive whole might not be a valid instance of the underlying language or formalism. For example, in an IDE, the respective artefact might show errors which makes this approach annoying at times. Because of its technical simplicity, the approach can be easily retrofitted to all kinds of artifacts: documentation, code, models.

≈≈≈

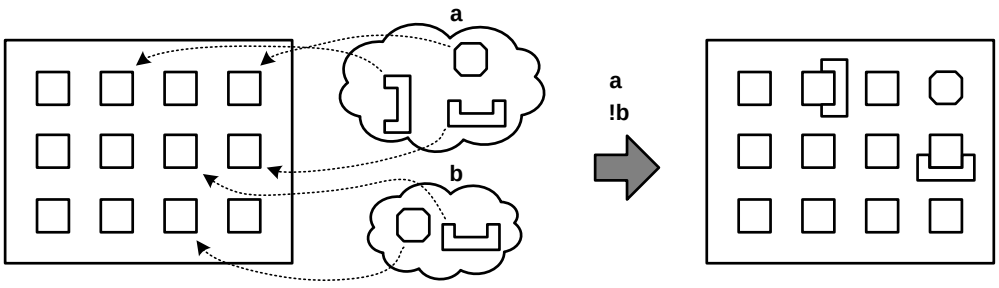
ifdefs in C and C++ are a well-known example of this strategy. A preprocessor removes all code regions, whose ifdef condition evaluates to false. When calling the compiler/preprocessor, you have to provide a number of symbols that are evaluated as part of the ifdef conditions.

Conditional compilation can also be found in other languages. Preprocessors that treat the source code simply as text are available for many languages and are part of many PLE tool suites, such as pure::variants [5] or Software Gears [6].

The Autosar [7] standard, as well as other modeling formalisms, support the annotation of model elements with conditions that serve the same purpose. The model element (and potentially all its children) are removed from the model if the condition evaluates to false.

## Injection (aka positive variability)

**Inject additions into a minimal core. The core does not know about the variability, the additions point to the place where they need to be added.**




A base artifact made of various parts (the small rectangles) exists. There is also variant specific code (the strange shapes), connected to features external to the actual artifact and pointing to the parts of the artifact to which they can be attached.

Defining a variant means that the variant specific code associated with the selected features are injected into the base artifact, to the parts they designated.

The clear advantage of this approach is that the core is typically small and contains only what is common for all products. The parts specific to a variant are kept external and added to the core only when necessary.

To be able to do this, however, there must be a way to refer to the location in the minimal core at which to add a variable part. This either requires the markup of hotspots or hooks in the minimal core or some way of pointing into the core from an external source. In the latter case, the core requires no modification and the approach can be used for implementing unexpected variability.

≈≈≈

Aspect Oriented Programming (AOP) [8] is a way to implement this strategy. Pointcuts are a way of selecting from a set of join points in the base asset. A joint point is an addressable location in the core. Instead of explicitly defining hooks, all instances of a specific language construct are automatically addressable.

Various preprocessors can be used in this way. However, they typically require the explicit markup of hooks in the minimal core.

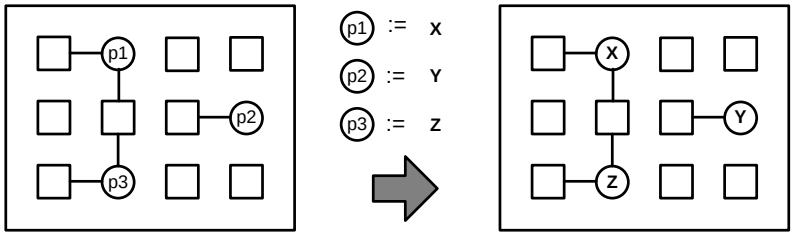
For models, injection is especially easy, since in most formalisms model elements are addressable by default. So it is possible to point to a model element, and add additional model elements to it, as long as the result is still a valid instance of the meta model.

The installation of optional packages for software systems is another example of this pattern.

An example in the architectural patterns world would be the Microkernel [9]. A microkernel-based system is one that provides a minimal set of functionality in its base functionality together with a protocol for plugging in additional pieces of functionality that makes use of the functionality in the microkernel, or other additions.

## Parameterization

**The artifact that shall be varied needs to define parameters. A variant is constructed by providing different values for those parameters. The parameters are usually typed to restrict the range of valid values.**




An artifact defines a number of (typed) parameters

A variant provides values for the parameters

The artifact that shall be parameterized needs to explicitly define the parameters, as well as a way to specify values (this makes this approach different from injection where it is possible to make it work without marking up the minimal whole). Hence, the variability is limited to the locations where parameters are defined. The core has to query the values of those parameters explicitly and use them for whatever it does. The approach requires the core to be explicitly aware and define all parameters, unexpected variability cannot be handled.

In most cases, the values for the parameters are relatively simple, such as strings, integers, booleans or regular expressions. However, in principle, they can be arbitrarily complex.

≈≈≈

A configuration file that is read out by the using application is a form of parameterization. The names of the parameters are predefined by the application, and when defining a variant, a set of values is supplied.

The strategy pattern is a form of parameterization, especially in combination with a factory. A variant is created by supplying an implementation of an

interface defined by the configurable application. Once again, the application has to explicitly query the factory, and the type of the values is defined by the interface which its strategy classes implement.

All kinds of other small, simple, or domain specific languages can be used as a form of parameterization. A scripting language in an application is a form of parameterization. That type of parameter is "valid program written in language X". Also, systems where some kind of behavior can be configured using workflow languages, activity diagrams, state machines or business rules is a form of parameterization. In this case, too, the languages used to define the behavior are the type of the parameter.

The classical approach of copying resources is also a form of parameterization. Consider the place where a logo is exchanged. The application defines a parameter ("logo for the company"), the type being the file type (such as GIF, 32x32 pixels) and the parameter is any valid image that makes sense as a logo for the company.

### Combinations

Of course there are also combinations of all of these approaches. Going back to the component example introduced in the CONSTRUCTION pattern, components that are wired together often also use PARAMETRIZATION to implement another, smaller grained form of variability.

Another combination is using PARAMETRIZATION to determine which parts are REMOVED or INJECTED.

## Acknowledgments

I want to thank my EuroPLoP 2009 workshop for the feedback on this paper: Christa Schwanninger, Klaus Marquardt, Didi Schütz, Rene Bredlau, Claudius Link and Ed Fernandez.

Thanks to Iris Groher for providing feedback on earlier version of this paper. I also want to thank my EuroPLoP 2009 shepherd Michael Stal for his repeated useful feedback.

## References

[1] http://www.softwareproductlines.com/introduction/introduction.html

[2] http://en.wikipedia.org/wiki/Feature_model

[3] Klaus Pohl, Günter Böckle, Frank van der Linden, Software Product Line Engineering. Foundations, Principles, and Techniques, http://www.amazon.de/Software-Engineering-Foundations-Principles-Techniques/dp/3540243720

[4] http://en.wikipedia.org/wiki/Domain-specific_language

[5] http://www.pure-systems.com/pure_variants.49.0.html

[6] http://www.biglever.com/solution/product.html

[7] http://www.autosar.org/

[8] http://en.wikipedia.org/wiki/Aspect-oriented_programming

[9] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, Pattern Oriented Software Architecture, Vol 1, http://www.amazon.com/Pattern-Oriented-Software-Architecture-System-Patterns/dp/0471958697