

Dietmar Schütz
Siemens AG, CT SE 2

Otto-Hahn-Ring 6
81739 München
Germany

eMail: dietmar.schuetz@siemens.com

Phone: +49 (89) 636-57380

Fax: +49 (89) 636-45450

COPING WITH VARIABILITY IN SOFTWARE DEVELOPMENT

Version 1.0, 22nd June. 2007, input for Writers Workshop at EuroPLoP2007

Summary

This collection of patterns operates in the realm of product line engineering. Variability is a key issue here, since it defines the boundary between the common parts (core assets, platform) of the product line, and the specific products derived from the core assets. The patterns describe how variability can be handled during the development process, and how it is implanted into the architecture of the core assets and hence all systems built on top of that.

Common Context

Software systems and software for systems that are developed following a product line engineering or a product family approach.

Example

Consider a company developing cellular phones. Typically, there are many different models, covering various market segments with distinct requirements. The overall set of features, varying between different models, contains look-up for phone numbers (a simple phone book, or a sophisticated address book), media player for different formats (mp3, wav), communication capabilities (single band, triple band, Bluetooth), camera with different resolution, voice control, etc.

In order to minimize development efforts for the different products, the development structure follows a product line engineering approach.

Throughout the following, two optional will serve as use cases to explain the concepts with practical background:

- A built-in camera
- A pluggable GPS-Module

Copyright granted to Hillside Europe for use at the EuroPLoP2007 conference.

Introduction

To avoid misconceptions, here are some definitions of the most important terms.

A *Product Line* is a set of different products in a common domain. If the products do not only share the domain, but a significant set of core assets they are built on, it's called a *Product Family*. The corresponding development approach is referred to as *Product Line Engineering (PLE)*. It is characterized by two distinct but interlinked development areas: *Domain Engineering*, focusing on the development of the core assets, and *Application Engineering*, dealing with the development of a single concrete product. One of the most important steps of domain engineering is the *Commonality/Variability-Analysis*, where all the common and different *Features* (high level functions that are visible to the user) are structured and assigned in order to characterize the family and distinguish the different products. Those features that are not common to all products define the *Variability Model*. It structures the whole set of *Variation Points*, things that may vary between products. *Variants* are concrete options that fit into the variation point.

Overview

A huge variety of techniques is applied actively in many projects. Examples range from approaches using configuration management (e.g. CM-Database copies) over build oriented concepts (shadows, replacement of header files containing #defines, search/replace in source files) to run time solutions (configuration files, hardware detection, flexibility patterns). The collection of patterns presented here yields at extracting the rationale behind those techniques, and organizing the underlying base concepts along different dimensions present in the context of product line engineering.

Patterns for basic concepts:

- Separated Variability Models
- Variation Point Discriminator
- Binding Time
- Variation Management and Control

Implementing variability takes places in different dimensions.

- Time perspective: decision time, resolution time (Early binding)
- Identification and control mechanisms
- Change mechanisms for refining a variant within the core assets.

After the basic concepts, some variation techniques are described in brief, highlighting the different dimensions mentioned above.

SEPARATED VARIABILITY MODELS

| | |
|------------------|--|
| Context | You have performed a Commonality/Variability-Analysis within your domain. The common features of your product family are identified, as well as the variation points and variants where single products differ from each other. A variability model provides support to describe distinct products by enabling optional features and selecting alternatives. |
| Problem | <p>The features reflect elements of the domain, and variability defines a structure around these. But when developing the core assets, architectural and design decisions follow other principles, such as “separation of concerns”, “coupling and cohesion”. The resulting structures do not necessarily fit the feature perspective. But nevertheless, the specific feature set of a concrete product controls the derivation process, and hence must be reflected in the core assets. How can you prevent the feature view from driving your architecture away from good engineering practice?</p> <p>The following forces influence the solution:</p> <ul style="list-style-type: none">• The usage of domain wording is essential to characterize the different products, and to reflect and stay connected to the customer needs• Implementation of a feature usually incorporates several components.• The variability model might contain restrictions regarding the combination of different features.• The implementation might add its own rules to that, thereby addressing implementation needs.• Valuable solutions (design and implementation fragments) from products developed in the past might be reused to build the core assets.• A stated need for variation is only a presumption of the future; it might (or will) change like usual requirements do when confronted with reality. |
| Solution | Clearly distinguish between the different perspectives on variability: use separate models to describe the variation in problem space and solution space. Introduce a mapping between both structures to derive the description of solution. |
| Structure | The problem space (user requirements) is best captured in a so-called feature tree. The nodes in this tree are associated with concrete features, or groups of them. Since the feature tree reflects the whole family, it must provide to express the variability too, which is done by annotations. Some nodes may be marked as optional, indicating a feature that may or may not be part of a |

concrete product. Other nodes allow a selection of one (or more) specific variants [Czarnecky]. Optional nodes or selection nodes represent variation points. In order to describe a concrete product, all variation points must be bound (decided on). In addition to combinatoric restrictions (usefulness, potential for simplifying implementation) (see [PureVariance]).

In solution space, the system decomposition (architecture and design) defines the structure of the variability tree. Variation points are realized as interfaces, with components as variants adhering to them (see [BigLever]).

The mapping between both structures establishes links between variation points on one side to those on the other side.

Note: there is not always a complete mapping in a way that every solution space variation point depends on one (or more) feature decisions. This is often the case for low level variation points (minor configuration parameters).

The mapping relations are not necessarily carved into a data structure that supports automated forward mapping. In environments that lack “production” character, indicated by small numbers of product instantiations, a manual mapping might be the better choice. But this does not free from explicitly engineering and documenting the mapping rules.

Dynamics

Working with these models is mainly characterised along the mapping structure in both directions:

- Forward Mapping from features to implementation.
- Backward Mapping from implementation to features.

Forward Mapping is the typical use case when initiating a new product from the product line. After deciding on all variation points in the feature model (problem space), the concrete choices are pushed through the mapping structure in order to derive the corresponding implementation variants. By applying these variants on the implementation variability model, the development structure for this concrete product is built up.

Backward Mapping from implementation to feature variability is used less often, typically in the early phases of the family development. The main purpose of this step is to reflect restrictions that arise within solution space back in the problem space. This prevents from offering choices on the feature level that are rated invalid after performing the forward mapping (error prevention instead of error detection).

The restrictions on the solution side can arise from concrete technical decisions, e.g. the number of slots for functional extension, or performance budgets. Furthermore, even abstract issues such as an excessive combinatoric complexity can be influenced on the feature level, e.g. by bundling features into packages that can be selected only as a whole.

Example Resolved The feature “Voice Recognition” is an option in the feature tree. In solution space, this feature influences several local options, such as the connector that hosts a line for the Push-To-Talk button, the memory that holds space for several recorded sounds to match against, the user interface that enables the “Assign voice command” menu entry for actions and phonebook entries.

Consequences *SEPARATED VARIABILITY MODELS* provides the **benefits** depicted below:

- *Functionnl fit.* In problem space as well as in solution space, the variability models reflect the specific needs of the development context. This provides an environment where variation represents concepts that are easily understood within their context.
- *Degree of freedom in architecture.* Architectural decisions are influenced by many (often conflicting) forces. A stakeholder less here provides room for positive effects in other areas.

On the other hand, the pattern carries the following **liabilities**:

- *Complex Mapping.* Most architectures follow horizontal separation paradigms, with a vertical perspective on functional aspects. This diversity is enhanced by grouping similarities in the different dimension. This usually leads to rather complex relationships when linking both models. The mapping structure requires careful engineering and maintenance.

Variant Depending on the granularity of the solution side model, it might be helpful to introduce an intermediate variability, acting as a high level model on the solution side.

VARIATION POINT DISCRIMINATOR

Problem Variation points are the key elements of variability models; they show up in all phases of product line engineering. Which properties and infrastructure are necessary to represent a variation point? Misconceptions in this area can cause tremendous maintenance effort in early development phases and artefacts, effecting later ones too.

The following **forces** influence the solution:

- Especially in the solution space, there are various methods to implement and resolve a variation point. All of these possibilities must be supported.
- During development, the variability model is subject to changes too, e.g. reorganisation the functional structure, change of binding time, or selecting a different implementation technique

- Variation points are related to others, to those within the same model as well as across the mapping structure.

Solution Use a consistent schema to identify and handle all variation points. Apply it on problem and solution space. Assign a unique name to each variation point. Associate it with a type that reflects the different possibilities for binding this variation point.

Structure A variation point consists of three core elements: its name, type, and value. Since the name is used to identify the variation point, it must be unique within in the context of the corresponding variability model.

Binding type: Boolean for optional features, an enumeration type for “one out of n” selectable features, and a set of enumeration values for “x out of n” selectable features.

Consequences The *VARIATION POINT DISCRIMINATOR* pattern provides the **benefits** depicted below:

- The *homogenous structure* of all variation points eases up building and changing the variability models, since all variation points provide the same interface to the “variability engineer”.
- This *common interface*, implemented e.g. as a base class, provides an easy way to realize the various kinds of relations.

On the other hand, the pattern carries the following **liabilities**:

- An *additional mapping* becomes necessary to bridge the gap between the homogenous discriminator and the different implantation schemes. For example, environment variables or directory names define their own namespace, and may come with additional restrictions that must be obeyed.
- This mapping and the resulting level of indirection can cause *performance drawbacks*, especially for variation points that are resolved at runtime, such as a file name to select a specific dll-file.

BINDING TIME

Common Problem Computers are (at least in theory) deterministic, hence when executing the program, all variability must be gone. When and how should variability be bound in the executables?

The following **forces** influence the solution:

- Early decisions reduce flexibility in later development phases.
- Late decisions allocate additional development resources:

- Runtime checks are expensive and increase the overall code size.
- Early binding might not be feasible due to lack of information, since information isn't available upfront (hardware detection, licence keys).
- It is considered good practice to reveal errors as early as possible.
- Some environments, especially development of safety critical systems, require 100% path coverage during test; hence "dead code" is futile.
- An implementation variant cannot be chosen before its functional counterparts, but maybe left "open" for other reasons.

Solution

Focus on the point in time when the variability is *resolved* best, after it has been *decided* on. Independent for each variation point, select the appropriate resolution time according to your needs: resolve as early as possible if you must care about resource and execution constraints on your target system, resolve late if you want to minimize development complexity. When the binding time is settled, select an appropriate control mechanism to implement the resolution.

Structure

Programs have to be developed before they can be executed. Hence, there are at least two different choices for binding a variation point: development time and execution time. The development time can be separated into compile time, when single source files are compiled to object files, and build time, when several objects are assembled into one ore more executables.

The following table characterizes different options for binding time, with early binding in the top and late binding at the bottom lines, and getting more detailed to the right. The bullet points provide some examples for specific techniques.

| | | |
|------------------|---|---|
| Development time | Compile time | Source selection time <ul style="list-style-type: none"> • <i>Config. Management</i> • <i>Directory Selection</i> • <i>File Selection</i> |
| | | Source compilation time <ul style="list-style-type: none"> • <i>Compile-time switches</i> • <i>Conditional compilation</i> • <i>Use of C++-templates</i> |
| | Build time <ul style="list-style-type: none"> • <i>make targets</i> • <i>resolving from libraries</i> | |

| | | |
|----------------|------------|--|
| Execution time | Start time | Cold start <ul style="list-style-type: none"> • <i>Configuration files</i> |
| | | Warm start <ul style="list-style-type: none"> • <i>Hardware detection</i> |
| | Run time | Dedicated State <ul style="list-style-type: none"> • <i>Reloading configuration tables</i> |
| | | On the Fly <ul style="list-style-type: none"> • <i>Various patterns, such as “Strategy” [GOF94] or “Microkernel” [POSA97]</i> |

Consequences The *BINDINGTIME* pattern provides the following **benefits**:

- *Minimal resources.* The resulting system has a minimal footprint, since it does not contain as less unnecessary (dead) code
- *Early detection of errors.* In the case of missing restrictions on solution variability model, there is a good chance to discover errors within a concrete set during development, or at least at sart up of the system, long before they can occur in the field and cause serious trouble.

On the other hand, the pattern carries the following **liabilities**:

- *Discriminator Access Method.* Since the discriminators are used all over the development phases and artefacts, every access point requires its own specific method to derive and use a discriminator value.

VARIATION MANAGEMENT AND CONTROL

Problem

Huge projects dealing with variability often end up using a big number of variation techniques (sometimes more then 20) within the same project. As a result, the values (variant names) characterising a specific product are spread across many different development artefacts. How can the different products (respectively the variation profiles describing them) be managed consistently?

The following **forces** influence the solution:

- Product and portfolio management recognizes a variation point as an abstract concept, disregarding details of implementation techniques.

This perspective supports consistent profiles that characterizes a specific product (with all variability bound)

- Different variation points have different binding times, and require different resolution or implementation mechanisms.
- Constraints on the “executable” might enforce specific techniques. For example, in safety critical environments, the installed artefacts must be protected with checksums, which is difficult to achieve for environment variables or registry entries.

Solution Split each variation points into its management and implementation view. Use an integrated management database as master, and derive the implementation relevant information from it, using automated change mechanisms.

Structure The *variability management database* serves as central point that contains the whole variability model, at least the list of all variation points and there possible variants. Furthermore, the database provides profiles for the defined products, each specifying the concrete variants that characterize the product. Even variation points that are decided at runtime (e.g: by looking up the hardware structure) should be listed here.

Depending on the resolution time and mechanism, parts of a profile are stored redundantly a different location, where they can be accessed for resolution. Examples are an environment variable that is resolved within a make-file when building a product, or a configuration file that is part of the installation package for a concrete product and read during start-up of the software.

Variation points that are resolved during build time are typically supported by a *change mechanism* that is used to adapt the generic elements of the core assets to the specific needs of a variant, e.g. by copying files that belong to the selected variant into the predefined build directory, or by modifying a makefile. Typical techniques are:

- Selection of objects for build (files, libraries, directories)
- Scripting (to modify the content of files)
- Generating (the content of) files

The latter techniques are also applicable for runtime variability, especially for adapting configuration files.

Consequences Applying the solution described above yields the following **benefits**:

- *Centralized access*. The whole set of variation points is accessible from a single point.

- *Freedom for binding.* There are no side effects that influence the decision on the binding resp. resolution mechanism.

On the other hand, the pattern carries the following **liabilities**:

- *Redundancy.* Storing most variation points twice requires measures to prevent inconsistencies. An optimal but not trivial approach here is an automated transition, using appropriate change mechanisms

COLLECTION OF TECHNIQUES

File selection on dll-linking, controlled by env vars, using get(env)

Compile and Build Time

- Setup Shadows
- Working areas
- Add/replace header files containing #defines
- Add/replace config files
- Linking in of “component” start-up functions (ModeManager)
- Text search and replace

Start-Up time

- HW detection
- Config settings in EEPROM or flash
- Config files

Run time

- Config files
- User input

RELATED INSIGHTS

VARIABILITY AND CONFIGUATION MANAGMENT (VARIATION OVER TIME)

Some publications on variability distinguish variation in space and variation in time. From my point of view, this is a little bit misleading, since both have almost nothing in common. Variation in time acts as a synonym for changes due to change requests or bug fixes that mostly have taken place in

the past. These are best managed as usual by means of a configuration management system.

But using such configuration management techniques (e.g. branches) for implementing the planned (future) variation in functionality (space) often ends up in a mess, with no easy way out. Root cause for this is excessive branching, since two crosscutting problem dimensions are forced on a common solution.

Credits

Thanks to my colleagues Horst Sauer and Jürgen Salecker for their patience in endless discussions. My shepherd Klaus Marquardt never gave up and eventually succeeded in pushing me back to the right path.

References

[GOF94]

E. Gamma, R. Helm, R. Johnson, J. Vlissides:

[POSA96]

F. Buschmann, R. Meunier, H. Rohnert, M. Stal, P. Sommerlad:
Pattern Oriented Software Architecture, A System of Patterns;
Wiley 1997

[BigLever]

www.biglever.com

[Marquardt]

Klaus Marquardt: xxx

[PureVariance]

www.pure-systems.com

[Kircher, Michael]

[Harrison, Neil]

[SPLE]

Klaus Pohl, Günter Böckle, Frank van der Linden: Software Product
Line Engineering; Springer 2005