Dietmar Schütz
Siemens AG, Corporate Technology
CT T DE IT1
System Architecture & Platforms

Otto-Hahn-Ring 6
81739 München
Germany

eMail: dietmar.schuetz@siemens.com
Phone: +49 (89) 636-57380
Fax: +49 (89) 636-45450

# *VARIABILITY REVERSE ENGINEERING*

Version 1.0, (Final Version for Printed Proceedings), EuroPLoP2009

Summary

In the realm of Product Line Engineering (PLE), Variability Management is one of the key issues. The success of the whole product line approach relies on the correctness of the variability models. Unfortunately, before transiting to PLE, knowledge on the variability is not addressed explicitly, but embedded in many development artefacts. This pattern provides an approach to extract that hidden knowledge, and transform it into the required problem side commonality/variability model.

Context

Product Line Engineering (PLE), Platform Development,
Product Business, Solution Business

An established development organisation with several successful similar projects has identified the potential for a Product Line (PL) approach. It has defined a business strategy and market scope to be covered, and developed a coarse roadmap and internal business case for developing reusable assets.

Example

Consider your company operates in solution business in the web applications domain, developing customer specific (software) applications. These applications typically share a common set of features and solutions. For every customer/application, a separate project is created, responsible to satisfy customer needs within the given budget.
In order to reduce their own development effort, the projects have applied a copy/paste approach on the code (and project) base, using the most similar project from the past as starting point for their own work.

This ad-hoc reuse has sped up initial development of other applications, but reveals weaknesses in an increasing number of maintenance scenarios. Every bug that is found has a high probability to affect other projects too, but is difficult to be located and fixed in the different branches.

Now, your management wants to establish a “platform” (better: product line engineering) approach in order to benefit from reusing common parts during software development and the entire software lifecycle. In the context of the “Commonality-Variability-Analysis” (C/V-analysis) tasks during domain analysis, the projects from the past are revisited in order to extract a useful set of commonalities that shape the basis for planned reuse in the future. In addition, the variant parts should be identified for a proper platform scoping, defining the complete set of core assets that should be provided upfront.

Problem

A lot of knowledge regarding commonality and variability that has been accumulated in the past, but is not explicitly documented. How can the undocumented knowledge be made accessible for future work, and contribute to a viable commonality/variability-model?

The following **forces** influence the solution:

- *Different kind of artefacts*.
  The exiting base of information from previous projects spans various types of artefacts (documents, specifications, tools, code). All of them might be a source for commonality and variability.
- *No explicit highlighting of variability*.
  From the perspective of a single project, variability does not exist, since the customer wants a *specific* system. Therefore, each project supports its own needs, but does not care about the differences to others. The copy/paste approach has helped to have a quick start, but couldn’t keep the different projects to follow their own, isolated path, resulting in an overly wide code base.
- *Constraints and dependencies*.
  Typically, a variability model does not only contain the possible variation points and variants, but also the dependencies between them, such as conflicting variants.

Solution

Extend the forward oriented variability modelling (feature based C/V-analysis) with backward oriented techniques (reverse engineering). To this end, analyse promising types of artefacts that have been created by previous development projects in order to extract candidates for problem side variability. Assess these candidates for there relevance to identify solution side variations points and variants. Map those back to (customer-visible) features in order to problem side variability model. With that model in place, assess the variability on both sides to derive technical constraints and dependencies as necessary part of the variability model.

**Structure**

The intended outcome of the C/V analysis is a *commonality/variability-model*. It contains features that are common to all products, and those where the products differentiate from others, characterized by variation points (e.g. colour) and specific variants (e.g. blue, green).
In order to distinguish this C/V model from the developmental solutions, this model is typically denominated as the *problem side C/V model*. It is counterparted by the *solution side C/V model*, which relates to components, modules, or code fragments of the realization structure and implementation. Both parts of the C/V-model are connected by a *mapping structure* that links the variations points from both sides together, hence allowing deriving a concrete implementation based on a given set of features.

Input sources for the analysis are all kinds of *artefacts* that have been created and/or modified in relation with the definition and development of concrete products. Since similar kinds of artefacts exist for the different products, they build a *comparison base* for the analysis.

**Realization**

Applying *VARIABILITY REVERSE ENGINEERING* incorporates at least the sequence of six steps described below.

1 *Decide on input sources.*
Based on the knowledge which are the key artefacts that your business and development operates on, establish a set of artefact types that probably will provide much to variability.

The starting point are definitely the artefacts available on the problem side:

- Marketing material containing product descriptions.
  These might even contain explicit variability information by means of (comparative) feature lists.
- User Manuals

On the solution side, typical promising artefact types are the “high level” specifications of the system:

- Requirements specifications
- Architecture specifications
- System test cases

Some others artefacts contain the variability information more directly, but maybe on a too fine level:

- Code
  (maybe even explicitly exposing variability, e.g. by means of conditional compilation directives)
- Configuration files (.ini-files, and similar)

Last but not least, the development environment provides structural information too,

- Configuration management structure (branches)
- Build/development (management) structures

2 *Compare to derive differences*.
For most kinds of artefacts, variants reveal themselves as differences between two documents.
The results are best if you do not compare two random documents to each other, but use the copied master and those derived from it, to limit the set of differences/changes to just one level. To this end, it might be useful to do some "project archaeology" and dig out these "based on" relationships. Unfortunately, they are not necessarily identical throughout the whole set of artefacts.

Comparing two files using text/line oriented tools like `diff` is nice for code, but there is also need for a semantic diff /compare. Sometimes this is provided by development tools. Otherwise, it might be helpful to export the artefacts to an xml representation and compare these files incorporating the hierarchical structure. But this approach needs careful observation, since for example the order of nodes in containers (not ordered lists) might lead to misinterpretation the results.

3 *Identify candidates*.
Walk through the different kinds of artefacts and the comparison findings, in order to identify common and variant parts. Different artefact types require different assessment techniques, related to their typical content and change scenarios that might have been applied during "copy/paste/modify" cycles.

Marketing documents:
Scan for keywords like "option", "additional", "alternative", since they are explicitly indicating variability. The features listed there can (after verification) directly added to the problem side C/V model, and used as input for considering variability. If there are comparative specs for the different products, they already contain most of the variability information.

Requirement documents:
Due to the "contractual" character of requirement documents, often polishing of wording is necessary. In order to not mistake these as variation candidates, it is helpful to look into related solution side artefacts (architecture and design documents, hopefully linked to the requirements) to assess the variability potential of the discovered changes.

Configuration management system:
This source of information might provide explicit variability candidates, e.g. by means of branches. Another useful source is the code and its changes over time. The changes (when taken from configuration management) often indicate the reason for change by check-in comments.

4 *Select solution side variability*.
After having all the differences and changes available, it is necessary to filter out the irrelevant elements. While this might be obvious for smaller code changes, all bigger differences must be reflected according to their

relevance to stakeholders (customer, product manager, key developer) regarding variation scenarios. The remaining elements are arranged into a solution side variability model, typically oriented along the structure of the system and development organisation, which contains the common parts too.

5 *Map back to problem side variability*.
We need to separate customer relevant requirements and variability from corresponding design related elements. For design related issues, try to analyze the rationale behind it –it could be an undocumented requirement or a constraint

- use established RE analysis techniques to extract the reason behind
- try to reverse the refinement step by abstraction/generalization

Use high level features identified in the solution domain as a guide for further reverse engineering activities and as completeness check

Finally merge all identified requirements, variability and constraints with requirements on solution side and consolidate the models.

6 *Complete white spots*.
When looking at your problem side, you might recognize that the system you see does not match the perspective of the customer. Use scoping techniques to define the boundaries of your product line, and complete missing elements within your desired scope.

Consequences *VARIABILITY REVERSE ENGINEERING* provides the **benefits** depicted below:

- *Efficiency*
  By focussing on artefacts types with high potential first (explicit variability, on problem side), the process of extracting variability the generates the most important results with minimal effort early (80:20-rule).

- *The variability model reflects knowledge from past projects*
  Even if not obvious upfront, the implicit variability knowledge from the past projects is incorporated into the commonality/variability-model, making it much more realistic than a strictly top down approach.

- *Derivation support*
  The artefact types that have provided useful content to the backward analysis are also promising candidates for forward oriented tasks: they should be supported by the platform for the efficient derivation of concrete projects/products.

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

- *Reverse Engineering can be laborious and expensive*
  Especially the tasks of deriving and assessing the candidates for variability can eat up tremendous resources, due to their sheer number.

- *Risk of overdoing*
  The past does not necessarily reflect the future. Hence focusing to much on the derived information can be misleading.

- *Risk of missing dependency information*
  Possible dependencies between variants (e.g. conflicts) are typically not expressed in the development artefacts, and hence cannot be derived from them. Hence, the variability model needs to be proof-read, consistency-checked and possibly extended afterwards.

Variants

In product business, there are even more sources: information gained from competitors can be subject to the variability reverse engineering too: products catalogues, (comparative) feature lists, reverse-engineered products. Although they do not necessarily fit to your solution side variability model (and therefore should be kept separately in the beginning), they valuably contribute to the problem side.

Credits

Thanks to my shepherd Hans Wegener, for his patience and guidance in busy times. To my colleagues Horst Sauer, Anne Hoffmann, and Christa Schwanninger, for providing their thoughts and never getting out of discussion. And last but not least I thank the participant of the writers workshop at EuroPLoP 2009, namely Alain-Georges Vouffo-Feudjio, Christa Schwanninger, Claudius Link, Ed Fernandez, Klaus Marquardt, Markus Völter, Michael Kircher, and Rene Bredlau.

References

[SPLE2006]
Pohl, Böckle, van der Linden, Software Product Line Engineering, Springer, 2005