Consolidating Customized Product Copies to Software Product Lines*

Benjamin Klatt, Klaus Krogmann
FZI Research Center for Information Technology
Haid-und-Neu-Str. 10-14,
76131 Karlsruhe, Germany
{klatt,krogmann}@fzi.de

Christian Wende
DevBoost GmbH
Erich-Ponto-Str. 19,
01097 Dresden, Germany
{christian.wende}@devboost.de

1 Introduction

Reusing existing software solutions as initial point for new projects is a frequent approach in software business. Copying existing code and adapting it to customer-specific needs allows for flexible and efficient software customization in the short term. But in the long term, a Software Product Line (SPL) approach with a single code base and explicitly managed variability reduces maintenance effort and eases instantiation of new products.

However, consolidating custom copies into an SPL afterwards, is not trivial and requires a lot of manual effort. For example, identifying relevant differences between customized copies requires to review a lot of code. State-of-the-art software difference analysis neither considers characteristics specific for copy-based customizations nor supports further interpretations of the differences found (e.g. relating thousands of low-level code changes). Furthermore, deriving a reasonable variability design requires experience and is not a software developer's everyday task.

In this paper, we present our product copy consolidation approach for software developers. It contributes i) a difference analysis adapted for code copy differencing, ii) a variability analysis to identify related differences, and iii) the derivation of a reasonable variability design.

2 Consolidation Process

As illustrated in Figure 1, consolidating customized product copies into a single-code-base SPL encompasses three main steps: Difference Analysis, Variability Design and the Consolidation Refactoring of the original implementations. These steps are related to typical tasks involved in software maintenance, but adapted to the specific needs of a consolidation.

As summarized by Pigoski [2](p. 6-4), developers spend 40%–60% of their maintenance effort on program comprehension, i.e. difference analysis in our approach. This is a major part of a consolidation process but it is also the least supported one.

In the following sections, we provide further details on the different steps of the consolidation process.

*Acknowledgment: This work was supported by the German Federal Ministry of Education and Research (BMBF), grant No. 01IS13023 A-C.

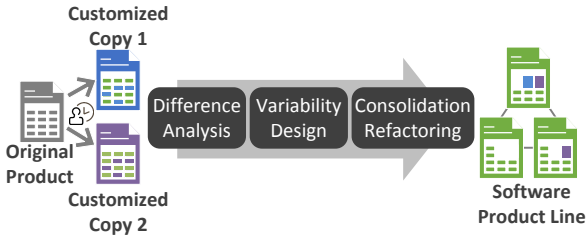

Figure 1: Consolidation Process

3 Difference Analysis

We have developed a customized difference analysis approach that is adapted for the needs for product-line consolidation in three directions: Respecting code structures, providing strict (Boolean) change classification, and respecting coding guidelines for copy-based customization if available.

Today's code comparison solutions do not always respect syntactic code structures. This leads to identified differences that might cut across two methods' bodies. In our approach, we detect differences on extracted syntax models. This allows to precisely identify changed software elements and detect relations between them later on.

Furthermore, we filter code elements not relevant for the software's behavior (e.g. code comments or layout information). However, we strictly detect any changes of elements in the scope and prefer false positively detected changes (i.e. they can be ignored later on) to avoid the loss of behavioral differences.

Coding-guidelines can include specific rules for code copying. For example, developers might be asked to introduce customer-specific suffixes to code unit names or introduce "extend"-relationships to the original code. Since these customization guidelines are vital for aligning different product copies, we also feed them into the difference analysis.

4 Variability Analysis

Having all differences detected, it is important to identify those related to each other. Related differences tend to contribute to the same customization and thus might need to be part of the same variant later on.

In our approach, we derive a Variation Point Model (VPM) from the differences detected before. The VPM contains variation points (VP), each referencing to a code location containing one of the differences.

At each VP, the code alternatives of the difference are referenced by variant elements.

Starting with this fine-grained model, we analyze the VPs to identify related ones and recommend reasonable aggregations. Recommending and applying aggregations is an iterative approach until the person responsible for the consolidation is satisfied with the VPs (i.e. the variability design). With each iteration, it is his decision to accept or decline the recommended aggregations. This allows him to consider organization aspects such as decisions to not consolidate specific code copies.

The variation point relationship analysis itself combines basic analyses, each able to identify a specific type of relationship (e.g. VP location, similar terms used in the code, common modifications or program dependencies). Based on the identified relationships, reasonable aggregations are recommended. Basic analyses can be individually combined to match project-specific needs (e.g. indicators for code belonging together).

5 Consolidation Refactoring

As a final step, the code copies' implementation must be transformed to a single code base according to the chosen variability design and selected variability realization techniques. Opposed to traditional refactorings (i.e. not changing the external behavior of software), consolidation refactorings might extend (i.e. change) the external behavior. The underlying goal of consolidation refactoring is to keep each individual variant/product copy functional. However, new functional combinations enabled by introducing variability are valid considered consolidation refactorings.

To implement consolidation refactorings, we are working on i) a refactoring method that explicitly distinguishes between introducing variability and restructuring code, and ii) specific refactoring automation to introduce variability mechanisms. The former focuses on guidelines and decision support. The latter is about novel refactoring specifications using well known formalization concepts, such as refactoring patterns described by Fowler et al. [3] or the refactoring role model defined by Reimann et al. [6]. Based on this formalization, we will automate the refactoring specifications to reduce the probability of errors compared to manual refactoring.

6 Existing Consolidation Approaches

SPLs and variability are established research topics nowadays. However, only a few existing approaches target the consolidation of customized code copies into an SPL with a single code base.

Rubin et al. [7] have developed a conceptual framework of how to merge customized product variants in general. They focus on a model level, but their general high-level algorithm matches to our approach.

In [8] Schütz presents a consolidation process, describes state-of-the-art capabilities and argues for the need of an automation as we target. In a similar way, others like Alves et al. [1], focus on refactoring existing SPLs, but also identified the lack of support for consolidating customized product copies and the necessity for automation.

Koschke et al. [5] presented an approach for consolidating customized product copies by assigning features to module structures and thus identifying differences between the customized copies. Their approach is complimentary to ours and could be used as an additional variability analysis if according module descriptions are available.

7 Prototype & Research Context

In our previous work [4], we presented the idea of tool support for evolutionary SPL development. Meanwhile, we are working on the integration with state-of-the-art development environments. Furthermore, in the project KoPL [1], we refine and enhance the approach for industrial applicability. This encompasses the adaptation of the analysis to be used by software developers in terms of required input and result presentation. Furthermore, extension points are introduced to support additional types of software artifacts, analyses and variability mechanisms.

Currently, a prototype of the analysis part is already available and evaluated with an open source case study based on ArgoUML-SPL and an industrial case study. The refactoring is in a design state and will be focused later in the project.

As lessons learned: A strong input of how desired SPL characteristics should look like (e.g. realization techniques or quality attributes) improves the approach. We call this an SPL Profile. Furthermore, the first step of "understanding" is the most crucial one for a consolidation.

References

[1] V. Alves, R. Gehyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena. Refactoring product lines. In *Proceedings of GPCE 2006*. ACM.

[2] P. Bourque and R. Dupuis. *Guide to the Software Engineering Body of Knowledge*. IEEE, 2004.

[3] M. Fowler, K. Beck, J. Brant, and W. Opdyke. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[4] B. Klatt and K. Krogmann. Towards Tool-Support for Evolutionary Software Product Line Development. In *Proceedings of WSR 2011*.

[5] R. Koschke, P. Frenzel, A. P. J. Breu, and K. Angstmann. Extending the reflexion method for consolidating software variants into product lines. *Software Quality Journal*, 2009.

[6] J. Reimann, M. Seifert, and U. Aß mann. On the reuse and recommendation of model refactoring specifications. *Software & Systems Modeling*, 12(3), 2012.

[7] J. Rubin and M. Chechik. A Framework for Managing Cloned Product Variants. In *Proceedings of ICSE 2013*. IEEE.

[8] D. Schütz. Variability Reverse Engineering. In *Proceedings of EuroPLoP 2009*.

[1] http://www.kopl-project.org