Multi-level Customization in Application Engineering

Krzysztof Czarnecki, Michał Antkiewicz, and Chang Hwan Peter Kim
University of Waterloo, Canada
Generative Software Development Lab
{kczarnec, mantkiew, chpkim}@swen.uwaterloo.ca
http://gp.uwaterloo.ca

Introduction

The goal of software product lines (SPLs) is to improve productivity, time-to-market, and quality of application development by leveraging the commonalities of systems within an application domain while managing their variations. SPLs package these commonalities and variations into a *domain-specific platform* (DSP), which may be customized through configuration settings or code extensions. Examples of large, vendor-provided DSPs are IBM's WebSphere Commerce for e-commerce applications and SAP's R/3 for enterprise resource management systems. The main advantage of creating a DSP is that its planned variability allows for a common product line architecture while its domain focus allows for domain-specific components that are functionality-rich.

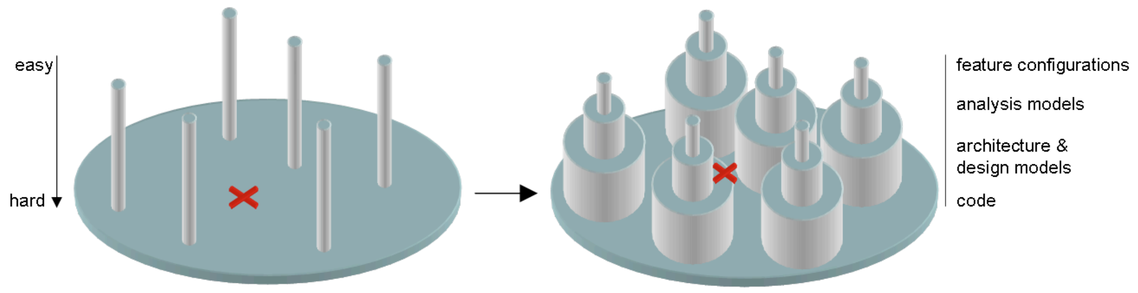

Figure 1: The customization cliff – problem and solution

Today's DSPs are usually implemented using code-centric technologies such as object-oriented frameworks, components, and sometimes even #ifdef preprocessor directives or similar macro-facilities, combined with wizards and form-based interfaces for easy configuration of requirements or *features*. Applications built on top of such DSPs represent some mixture of platform configuration settings and custom code. The left side of Figure 1 shows a typical DSP, where the horizontal dimension represents the scope of applications that can be derived using the DSP and the vertical dimension represents the easiness of deriving the applications. Cylinders represent applications that can be derived purely through feature configuration, while the rest of the applications on the plane, including the red cross, require custom coding. Custom coding is required if the customer desiring the system indicated by the red cross is not willing to accept any of the next-best purely configured solutions, i.e. the closest cylinder to the red cross. Unfortunately, in most DSPs, the scope covered by configuration is relatively sparse, and the transition from configuration to custom coding is rather abrupt, requiring a jump off the top of a cylinder. Steve Cook refers to this situation as the *customization cliff*[1]. The idea is that beyond simple configuration facilities such as wizards, which are usually provided to lower the initial entry barrier, the platform user faces custom coding against the gory details of the platform's application programming interface (API). For example, a change to the business workflow of an application generated from a particular feature configuration may require custom code that needs to interact with different parts of the DSP's API. Writing such code may require substantial effort since the relevant parts of the low-level API need to be learned and

[1] http://blogs.msdn.com/stevecook

used correctly, and the amount of code that needs to be written may also be substantial. This idea is at odds with the intuitive principle that “easy things should be easy to do and progressively more complex tasks should only get progressively harder to do, not insurmountably harder.”[2]

Model-driven software development [MDSD, Greenfield04] has the potential to eliminate the customization cliff in traditional, code-centric implementations of SPLs by offering multi-level modeling and customization as a middle ground between configuration and custom coding, as shown in the right part of Figure 1. In a nutshell, a complex code customization is replaced by more manageable customizations at multiple levels of abstraction. Furthermore, customizations at lower abstraction levels cover an increasing amount of the DSP scope. For example, changing the business workflow of our sample application may be done at the level of the analysis model with no need for custom coding, while interfacing to another application may require custom coding.

In the rest of the paper, after illustrating the idea of multi-level modeling of DSP-based applications with a concrete example, we compare and classify customization approaches that can be applied at different levels of abstraction and discuss the tradeoffs among them.

Multi-Level Modeling for DSP-Based Applications

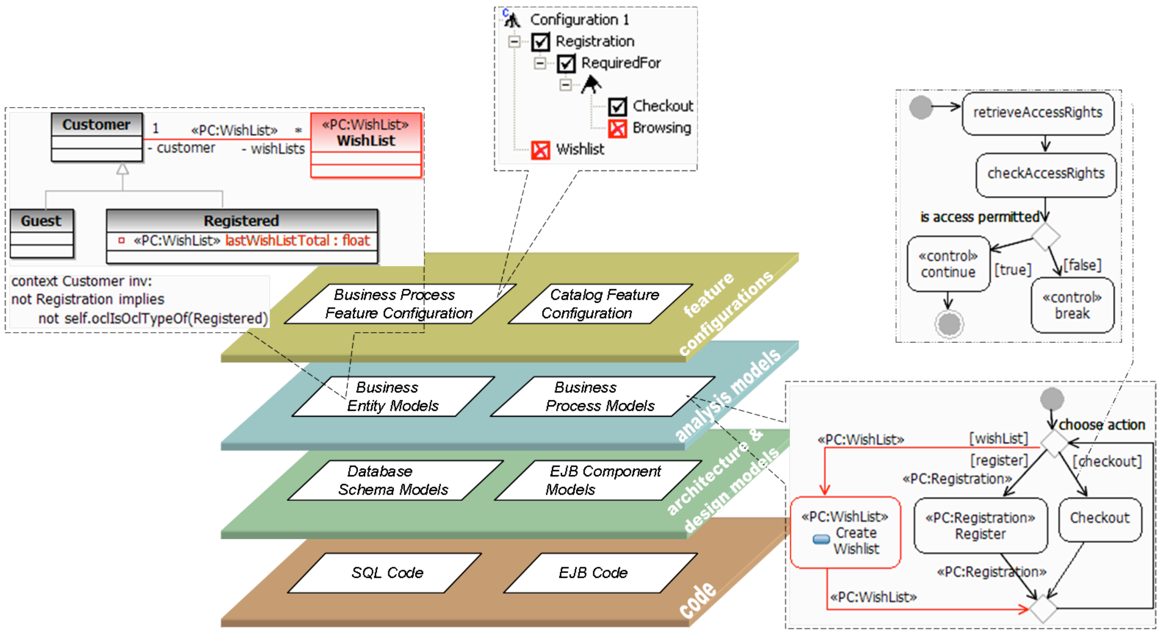

Figure 2: Multi-level representation and customization of an e-commerce system

The idea of a DSP supporting application engineering with multi-level customization is illustrated in Figure 2. We use an e-commerce DSP as an example. An application engineer first selects or *configures* the desired business process features, as shown in the top callout. Features available for selection are defined in a *feature model* [Kang90], which arranges the features in a composition-like hierarchy with some extra constraints. Feature configuration process may be guided through constraint-based facilities, such as consistency checking, choice propagation, and auto-completion [Batory05, Czarnecki05b]. For example, in the top callout, selecting `Checkout` would cause the automatic selection of the undecided features `Registration` and `RequiredFor` due to choice propagation.

Based on the feature selection, a default e-commerce application with requirements and design models and code is generated. But for this generation to be possible, feature model must be mapped to generic requirements and design models. One way to achieve such mappings, called *feature-based model templates*, is to annotate model elements with *presence conditions*, which are Boolean formulas over features [Czarnecki05a, Czarnecki06b]. For a given feature configuration, the presence conditions are

[2] http://blogs.msdn.com/garethj

evaluated and the elements with false conditions are removed. In Figure 2, since `WishList` feature was eliminated, `Create WishList` action, `WishList` class and its only association, and `lastWishListTotal` attribute of `Registered` class are eliminated (as indicated by the red color), thereby generating models from model templates with annotations.

Another way, called *feature-based restriction*, is to constrain models based on a feature configuration [Wasowski04, Czarnecki06]. In Figure 2, an OCL constraint below the class diagram states that if `Registration` feature has been eliminated, then instances of `Customer` class cannot be direct instances of `Registered`, which effectively eliminates `Registered`. Since `Registration` feature has been selected, the constraint has no effect. A comparison between feature-based model templates and feature-based restriction is given in [Czarnecki06].

Yet another way of mapping features to models is using *aspects*. In Figure 2, the top-right callout shows an authentication activity, which is part of a security aspect. A separate rule-based access policy specifies locations in business processes, such as before the `Checkout` action, where the authentication activity has to be woven.

The default implementation and models that were generated for the particular feature configuration most probably need to be further customized through extensions or modifications at multiple levels. Changes to one level need to be propagated to all other levels or at least the ones below the changed level. Both generation and change propagation are achieved using the discussed feature-to-model mappings, as well as model-to-model and model-to-code mappings, which are studied extensively in the field of model transformation (see [Czarnecki06a] for a survey on this topic).

Classification of Customization Approaches

Up to this point, multi-level customization has been discussed in the context of mapping between levels, without addressing how customization at each level occurs. In this section, we discuss a variety of customization approaches, which are classified in Table 1. Customization approaches can firstly be divided according to the modeling approach.

Table 1 Classification of Customization Approaches

Modeling approach	**Customization access to lower levels**	**Customizations distinguishable from their host layer or not**	**Special implementation technology required**
Black-box	None	Indistinguishable	None
	Indirect via mark-up	Distinguishable	Mark-ups
	Indirect via escapes	Distinguishable	Escapes
Multi-level	Direct without round-trip	Indistinguishable	None
		Distinguishable	Protected regions
			API-based, e.g., through sub-classing, call to/from customization, and partial classes
			Aspects
	Direct with round-trip	Indistinguishable	Analysis and transformation
		Distinguishable	

Black-box modeling approach

The black-box modeling approach is analogous to using a model compiler, which exposes only the top level model for customization. We include the black-box approach in our classification as a degenerate case of multi-level customization for the sake of completeness. The black-box approach is effective if all desired systems are expressible in the source modeling language. Platform specific markups, which are annotations for controlling source-to-target transformations [MDSD], offer limited customization. Escapes, which are fragments in the target notation embedded in the source, enable more customization. For example, in a modeling tool like Rational Rose RealTime, one can embed C++ code for actions in statecharts. Both markups and escapes allow only indirect and limited (i.e. preplanned) access to the lower level for customization. A notable drawback of both markups and escapes is that they mix levels of abstraction by directly placing lower-level customizations at a higher level. Comparatively, escapes are more problematic than markups in this respect since a source model can be considered as complete without the markups but not without the escapes.

Multi-level modeling approach

Multi-level modeling affords direct editing of the lower levels for customization. Multi-level modeling may or may not support round-tripping, which propagates lower level changes to the higher level and higher-level changes to the entire lower level including the customizations. In current practice, most multi-level modeling tools do not offer round-tripping. Another important dimension is whether the customized parts of a layer can be distinguished from the rest of the layer. Indistinguishable customizations without round-tripping are not practical since the customizations will be lost after regeneration. Indeed, current multi-level modeling tools commonly support distinguishable customizations without round-tripping. There are numerous ways to distinguish customizations. A protected region specially marks a customization to prevent regeneration from overriding it. This marking may be done in an implicit way through machinery that keeps track of user's edits behind the scenes, as in CompuWare's OptimalJ. Another approach is to separate the customization completely from the generated parts using mechanisms of the modeling or programming language. Such API-centric techniques include subclassing generated classes, invoking generated elements by customized elements or vice versa, and using C#'s partial classes [Greenfield04]. Yet another approach is to separate customizations into aspects. Protected regions and aspects facilitate unplanned customization since they allow any part of the generated model or code to be changed, unlike API-centric techniques. Although rare due to its difficulty, round-tripping can be achieved by defining a reverse and a forward mapping between two levels that identify the higher level concepts in the lower level and transforms the higher level concepts into the lower level elements respectively. Customization is typically indistinguishable from generated elements in round-tripping, but distinction may conceivably be leveraged to perform more sophisticated round-tripping.

The mappings between features and models discussed previously support any kind of direct customization without round-tripping. We are not aware of any comprehensive round-tripping support in feature-to-model mappings, but it is conceivably similar to model-to-code round-tripping in Framework-Specific Modeling Languages [Antkiewicz06].

Conclusion

In this paper we introduced the customization cliff – a situation in which the application engineers face custom coding after initial product configuration. We discussed the reasons behind the customization cliff and proposed multi-level customization as the solution. In multi-level customization, the initial product configuration results in a number of models and code at different levels of abstraction, enabling the engineers to apply customizations at appropriate levels. We also discussed the need for models to be connected by various kinds of mappings to enable change propagation between models and code. In particular, we discussed mappings between feature models and other models which are specific to SPLs. We also classified different customization approaches ranging from black-box to multi-level approaches and discussed their properties and tradeoffs.

Multi-level customization is an existing technique whose value has already been demonstrated through industrial support, including OptimalJ. However, multi-level customization involving features has not been explored in practice yet. Although a larger example in the context of a realistic e-commerce DSP within our group demonstrated the scalability of feature-based model templates [Lau06], realizing the vision of multi-level customization from features to models to code remains a future work item that boasts many challenges. In particular, multi-level round-tripping, verifying product derivation correctness, and migrating customizations back into the DSP offer exciting opportunities for future exploration.

References

[Antkiewicz06] M. Antkiewicz and K. Czarnecki. *Framework-specific Modeling Languages with Round-trip Engineering.* To appear in O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, Proceedings of Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genoa, Italy, October 2006.

[Batory03] D. Batory, J. N. Sarvela, and A. Rauschmayer. *Scaling stepwise refinement.* In Proceedings of the 25th International Conference on Software Engineering (ICSE), Portland, Oregon, pages 187–197, Los Alamitos, CA, 2003. IEEE Computer Society.

[Batory05] D. S. Batory. *Feature models, grammars, and propositional formulas.* In J. H. Obbink and K. Pohl, editors, Software Product Lines, 9th International Conference, SPLC 2005, Rennes, France, September 26-29, 2005, Proceedings, volume 3714 of Lecture Notes in Computer Science, pages 7–20. Springer, 2005.

[Czarnecki05a] K. Czarnecki, M. Antkiewicz. *Mapping features to models: A template approach based on superimposed variants*. In R. Glueck and M. Lowry, editors, GPCE 2005 - Generative Programming and Component Engineering. 4th International Conference, Tallinn, Estonia, Sept. 29 – Oct. 1, 2005, Proceedings, volume 3676 of LNCS, pages 422 – 437, Springer, 2005.

[Czarnecki05b] K. Czarnecki and C. H. P. Kim. Cardinality-based feature modeling and constraints: a progress report. In *International Workshop on Software Factories*, San Diego, California, Oct 2005. Paper available from http://softwarefactories.com/workshops/OOPSLA-2005/Papers/Czarnecki.pdf.

[Czarnecki06] K. Czarnecki, C. H. P. Kim and K. T. Kalleberg. *Feature Models are Views on Ontologies.* In Software Product Line Conference (SPLC), Baltimore, USA, August 21-24, 2006, IEEE CS, 2006.

[Czarnecki06a] K. Czarnecki and S. Helsen. *A Characterization and Categorization of Model Transformation Approaches.* IBM Systems Journal, 2006.

[Czarnecki06b] K. Czarnecki, K. Pietroszek. *Verifying Feature-Based Model Templates Against OCL Well-Formedness Constraints.* In Generative Programming and Component Engineering (GPCE), 2006.

[Greenfield04] J. Greenfield and K. Short. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, Indianapolis, IN, 2004.

[Kang90] K. Kang, S. Cohen, J. Hess, W. Novak, A. Peterson. *Feature-oriented domain analysis (FODA) feasibility study*. Technical Report, Carnegie Mellon University, Software Engineering Institute, CMU/SE-90-TR-21, 1990.

[Lau06] S. Q. Lau. *Domain Analysis of E-commerce Systems Using Feature-Based Model Templates.* MASc thesis, University of Waterloo, Ontario, Canada, Jan. 2006. Available from http://gp.uwaterloo.ca/.

[MDSD] T. Stahl, M. Völter. *Model-Driven Software Development: Technology, Engineering, Management.* Wiley, 2006.

[Wasowski04] A. Wasowski. *Automatic generation of program families by model restrictions.* In R. L. Nord, editor, Software Product Lines: Third International Conference, SPLC 2004, Boston, MA, USA, August 30-September 2, 2004. Proceedings, volume 3154 of Lecture Notes in Computer Science, pages 73–89, Heidelberg, Germany, 2004. Springer-Verlag.