Journal of Universal Computer Science, vol. 16, no. 17 (2010), 2313-2332
submitted: 15/2/10, accepted: 30/8/10, appeared: 1/9/10 © *J.UCS*

UML Support for Designing Software Product Lines: The Package Merge Mechanism

Miguel A. Laguna
(Department of Computer Science, University of Valladolid, Spain
mlaguna@infor.uva.es)

José M. Marqués
(Department of Computer Science, University of Valladolid, Spain
jmcc@infor.uva.es)

Abstract: Software product lines have become a successful but challenging approach to software reuse. Some of the problems that hinder the adoption of this development paradigm are the conceptual gap between the variability and design models, as well as the complexity of the traceability management between them. Most current development methods use UML stereotypes or modify UML to face variability and traceability issues. Commercial tools focus mainly on code management, at a fine-grained level. However, the use of specialized techniques and tools represent additional barriers for the widespread introduction of product lines in software companies. In this paper, we propose an alternative based on the UML *package merge* mechanisms to reflect the structure of the variability models in product line *package architecture*, thus making the traceability of the configuration decisions straightforward. This package architecture and the configuration of the concrete products are automatically generated (using Model Driven Engineering techniques) from the variability models. As an additional advantage, the package merge mechanism can be directly implemented at code level using partial classes (present in languages such as C#). To support the proposal, we have developed a tool incorporated into MS Visual Studio. This tool permits the product line variability to be modeled and the required transformations to be automated, including the final compilation of concrete products. A case study of a successful experience is described in the article as an example of applying these techniques and tools. The proposed approach, a combination of UML techniques and conventional IDE tools, can make the development of product lines easier for an organization as it removes the need for specialized tools and personnel.

Keywords: UML, Merge Relationship, Software Product Line, Variability, Traceability.
Categories: D.2.2, D.2.13

1 Introduction

Software product lines (SPL) are probably the most successful approach in the field of industrial software reuse, due to the combination of a systematic development and the reuse of coarse-grained components [Bosch, 00], [Clements, 01]. However, the approach is complex and requires a great effort for the companies that address it [Bosch, 00]. Our approach tries to adhere to de facto industrial standards whenever possible. For example, we incorporate specific techniques of *Product Line Engineering* in a process parallel to the conventional *Application Engineering*

[Laguna, 03]. This article focuses on the way that standard UML can be used to deal with the particularities of SPL design.

The distinctive characteristics of SPL development are variability and traceability management. There is wide agreement about expressing SPL variability (and commonality) by means of a feature model in some of their multiple versions such as FODA [Kang, 90], FORM [Kang, 98], or that proposed in [Czarnecki, 05b], which is the one used in this paper. A feature is defined as a characteristic relevant for a stakeholder that represents a common or variable aspect of a product [Kang, 90]. The technique is used to capture an SPL domain analysis (with mandatory, optional and alternative features) and to configure each SPL specific product.

The SPL is usually designed using UML models, taking into account the variability and commonality requirements. Feature and UML models must be used in combination: we must connect the variable (i.e., optional or alternative) features, through traceability links, with the related variation points of the design models that implement the product line. The explicit connection between feature and UML models allows the instantiation of the product line in each specific product to be automated. However, this apparently simple activity is far from trivial due to the complexity of the SPL variability and traceability management in the design/code levels [Sochos, 04]. The usual solutions found in the literature introduce modifications in the UML meta-model or add specific stereotypes (see the Related Work Section for a more detailed discussion). Contrary to these options, we propose to maintain the SPL design models within the standards, using the *package merge* mechanism of UML 2 [OMG, 03]. We describe the common and variable parts of the SPL (logical or physical) architecture using a UML package diagram [Hofmeister, 99], the *SPL package architecture*, with the packages connected by merge relationships. Two transformations generate the SPL package architecture (including the traceability links), and the configuration of the concrete products. Figure 1 shows schematically the relations between the SPL models (SPL feature model and SPL package architecture) and the derived product models (feature configuration and derived product architecture). Note that the instantiation of a specific product is achieved in two steps: feature model configuration and the automatic config2product transformation. The *package merge* concept is language neutral and can be implemented using C# partial classes or similar Java mechanisms. The C# option is supported by a tool for SPL modeling and configuration, integrated in MS Visual Studio. The tool implements the transformations that generate the SPL package architecture and the compiler instructions for each SPL product configuration.

The rest of the article is organized as follows: Section 2 presents the proposal: the requirements that define the problem, the details of the solution, based on the UML *package merge* mechanism, and the definition of the transformations that generate the structure of the SPL design and the package configuration of the SPL concrete products, starting from the feature models. Section 3 introduces the Feature Modeling Tool (FMT), designed to support the proposed techniques and transformations, and incorporated natively into the MS Visual Studio development platform. Section 4 describes a case study of an e-Commerce SPL where the proposal has been applied. In Section 5, the related work is analyzed and, finally, Sections 6 and 7 conclude the article and outline future work.

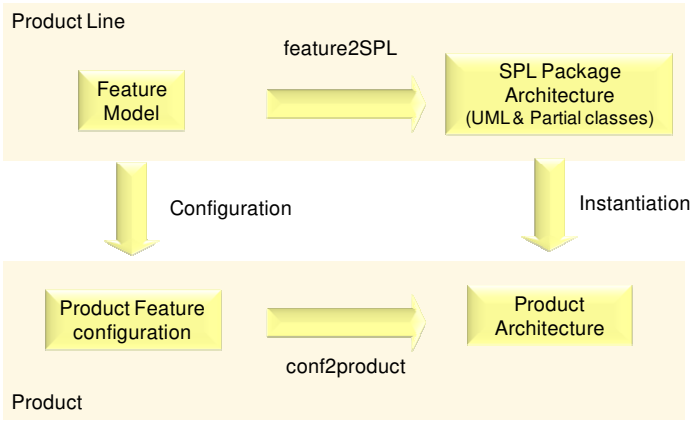

Figure 1: A global vision of the proposed integration of UML with feature models and transformation techniques to manage SPL variability

2 Design of Software Product Lines and Standard UML

The design of a software product line must reflect the common and variable parts of the diverse products that can be derived from the SPL generic solution. This derivation basically consists of two steps: a) select the desired features of a concrete product and, b) follow the traceability links to automatically instantiate that product. Before presenting our proposal, Subsection 2.1 states the problems that must be solved to apply step b. Then, Subsection 2.2 shows how the *package merge* mechanism of UML 2 provides an efficient technique to model the common and variable aspects of the SPL design. Finally, Section 2.3 explains how that SPL model and the associated traceability links can be automatically generated from the feature models.

2.1 Statement of the Problem: UML Variability and Feature Traceability

The UML diagrams have several mechanisms to express variability but these variants refer to specific products and not to product lines. For example, the use cases <<extend>> relationship allows different payment types to be modeled in a typical sales system. All the payment types can be valid in execution time and will therefore be present in each specific solution or, maybe, different applications of the product line will only admit some subsets of payment types. We need different mechanisms to express the variability in the specific product and in the product line levels. However, in the seminal work of Jacobson [Jacobson, 97], there was no fundamental difference between the representation of the variability at either level and the <<extend>> relationship is one of the pillars which supports his method. This approach lacks any separation between the representation of the variants of the product line and the specific variability of each final product (see Figure 2: an SPL use case diagram has no difference with a conventional one).

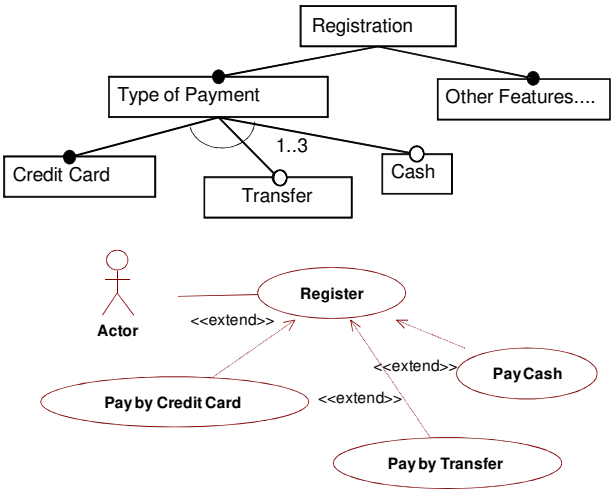

Figure 2: A fragment of a typical SPL feature model (hollow circles represent optional features while filled circles represent mandatory ones), and a use case diagram that applies the extend use case mechanism to represent SPL variability

Similar considerations can be made about the rest of the static or dynamic UML models. In a class diagram, a specialization of the concept *payment* can suppose several valid payment types at the moment of the execution of a system or the superset of all the possibilities contemplated in a sales domain corresponding to the whole product line. Figure 3 depicts a fragment of an SPL class diagram and two possible products. To deal with this problem, several solutions based on UML stereotypes to differentiate SPL from product models have been proposed (see Section 5 for examples as in [Gomaa, 04]).

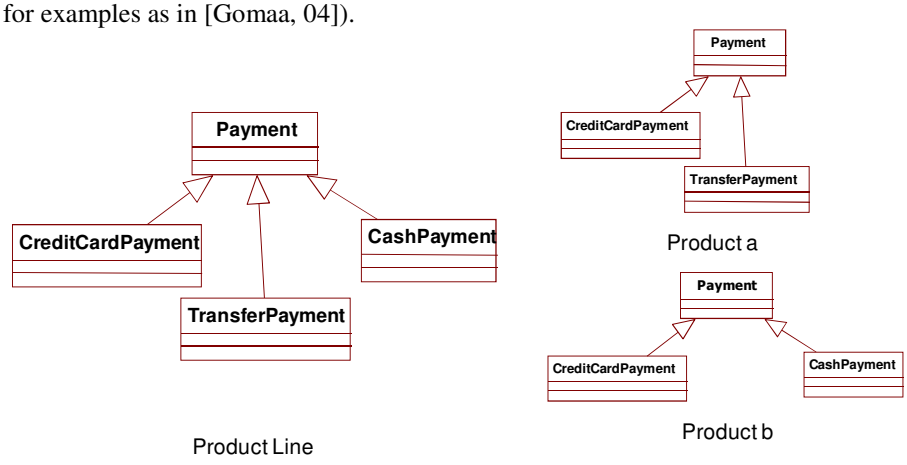

Figure 3: A fragment of an SPL class diagram and two possible derived products

On the other hand, the variants in UML models can be distributed between several modeling elements. The problem is due to the fact that a variable feature can correspond to several elements in a UML model. For example, selecting a feature "*Credit Card*" implies that, at least one class *CreditCardPayment* (with attributes *cardNumber*, *validityDate*, *authorizationNumber*, etc.), and a set of associations with other involved classes (*Bank*, *Customer*, *Receipt*…), will appear in the static model, in addition to several use cases and the related sequence diagrams). We must therefore assign the traceability relationship between elements of the two levels with a "one-to-many" multiplicity, involving many diverse UML elements in several models. This problem is not well addressed by the simple solutions based on stereotypes where an optional feature is expected to relate to a <<variant>> stereotyped class or method. A more detailed discussion about these problems and the different proposed solutions was previously presented in [Laguna, 07]. As a result of that study, we established a minimum set of requirements that a useful technique of representation, selection and management of the variability in the design level must accomplish:

1. The technique must allow all the variations associated to each variable feature to be located in one part of the model in order to facilitate the management of the traceability. For example, a feature that originates two classes and one association relationship that should be in the same package (the UML grouping construct).
2. The technique must discriminate between the variability of the product line and the intrinsic variability of the specific products.
3. The technique must maintain unchanged the UML meta-model, removing the entry barrier to this development approach for the majority of engineers, and additionally allowing the use of conventional CASE tools.

Related with these requirements, it would be a remarkable enhancement if the selected mechanism could have continuity from requirements (including feature models) until the implementation models, shifting toward the scheme of *"seamless development"* of the object-oriented principles. In the following Subsections, a solution that fulfills the previous requirements, based on the *package merge* mechanism of UML 2, is explained in detail.

2.2 The Package Merge Mechanism and SPL Design

To solve the problems set out in the previous Subsection, we aim to represent the variability in UML models using the package merge mechanism, defined in the UML 2 infrastructure meta-model and used exhaustively in the definition of UML 2 [OMG, 03]. The package merge mechanism basically consists of adding details to the models in an incremental way. According to the specification of UML 2, <<merge>>> is defined as a relationship between two packages that indicates that the contents of both are combined. It is very similar to the generalization and is used when elements in different packages have the same name and represent the same concept, beginning with a common base. The concept is extended incrementally in each separate package. Selecting the desired packages, it is possible to obtain a tailored definition from all the possible ones. Though we focus on class diagrams in this work, the mechanism can be extended to UML behavior models (in particular use cases, state charts, and sequence diagrams). Evidently, the rules that establish the specification of

UML 2 are strict so as to avoid inconsistencies. For example, cycles are not allowed, the resulting multiplicity ranges are the least restrictive possible, the operations should conform in number, order and type of parameters, etc.

The application of this mechanism to our problem consists of establishing an initial package for each UML model that embodies the common part of the product line. Then, associated to each variable feature, a new package with the same name is added, so that all the necessary changes in the model remain located in it, fulfilling one of the requirements of the previous Subsection. This package is connected through a <<merge>> relationship with its base package, that is, the package associated with the parent feature. The sense of the relationship expresses the dependence between packages: the base or merged package can always be included in a specific product, the receiving package is an extension of the base package and can only be included if the base package is also selected. This is exactly how the expert decides which features are included during the configuration process.

Each package can be organized in the ordinary way (using package composition and <<import>> or <<access>> relationships) and can contain the packages that reflect the interface and persistence layers, the details of a tiered architecture, or dynamic UML models. The result is a specific model, the SPL *package architecture,* which shows the global organization of the product line artifacts.

In the example of the use case packages of Figure 4 (depicting an oversimplified product catalog SPL), one or two packages could be added to the base package, obtaining three possible products in the product line. Notice that mandatory features do not generate additional packages. As they are always included in the product configuration with their parents, the associated UML elements are included in the existing package.

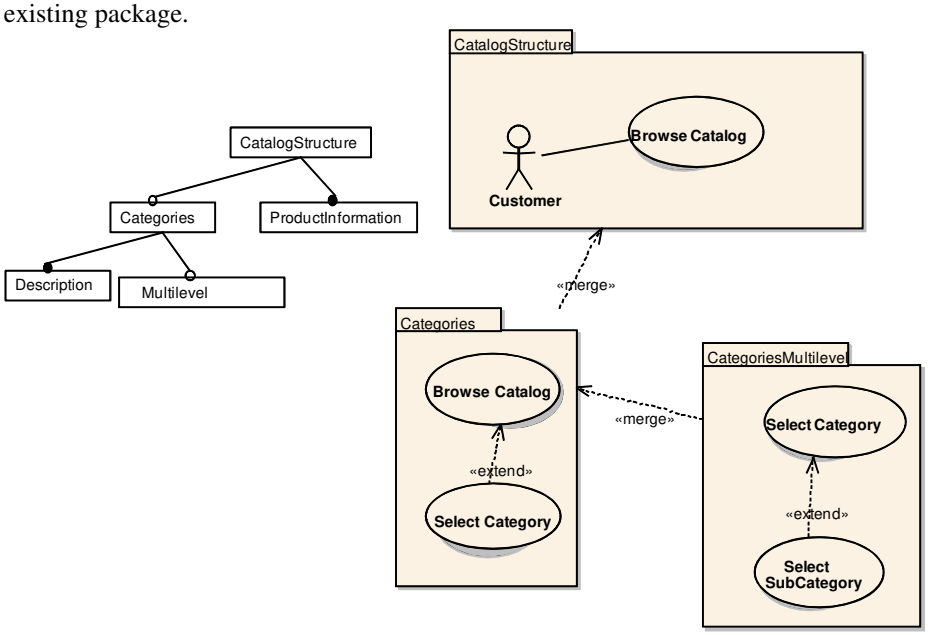

Figure 4: Example of the package merge mechanism applied to use case packages

Though use cases are very useful to describe the product line requirements, the static models are the preferred artifacts, since they represent the most useful views of SPL design and implementation. In Figure 5, the connection between the structure of the feature model and the corresponding structure of the packages can be appreciated. Each variability point detected in the feature model originates a package that may or may not be combined in product development time, according to the selected configuration. In Figure 5 it can be seen how a mandatory catalog feature in the product line has, optionally, a basic categories structure (package *Categories*). If present, each category has at least a description and each product has a zero or one associated category (a new class *Category,* an attribute and two associations are needed). To add the *CategoriesMultilevel* optional feature, it is necessary to add a reflexive association to the class *Category*, as shown in the corresponding package. Here, we can see how an optional feature is translated, not into a specialized class, as the stereotype approaches suggested, but simply into an association. In our approach, if (during the selection/configuration process) the *Categories* and the *CategoriesMultilevel* features are selected for a specific product, the traceability links add the corresponding packages to the design model of the product. The ordered merge of these packages originates the final solution. This approach solves the requirements 1, 2, and 3, stated in Subsection 2.1:

1. The technique allows all the variations associated to each variable feature to be located in one package.
2. The technique separates the variability of the product line (by means of the package merge) from the intrinsic variability of the specific products.
3. The technique maintains the UML meta-model unchanged.

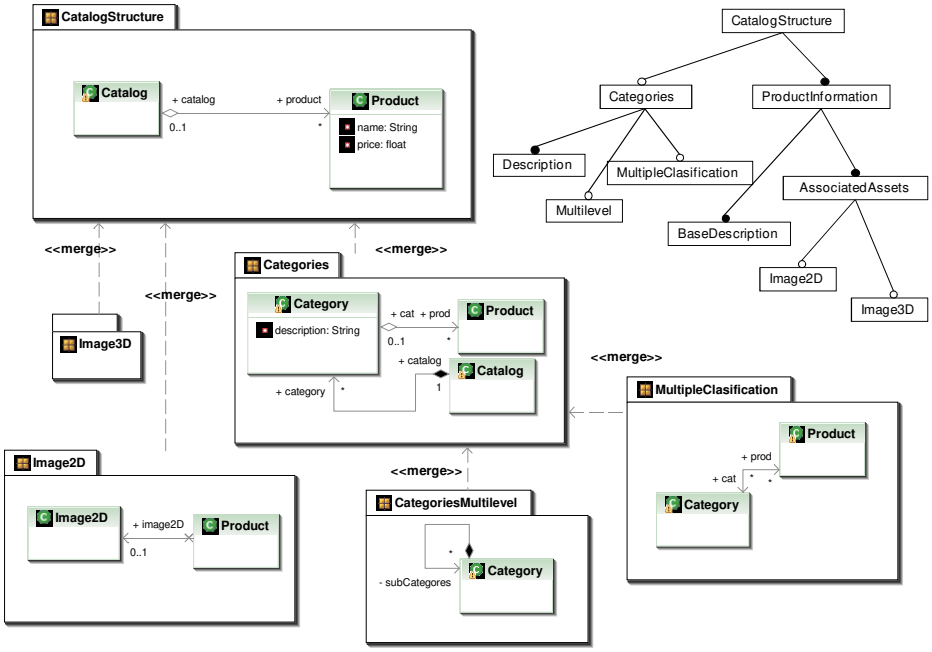

Figure 5: A partial Feature Model and a possible UML design

To achieve the goal of seamless development, the traceability links must include the implementation models. For this, we propose to split the code into fragments included in the SPL packages. The use of *mixins* (defined as fragments of classes that are intended to be composed with others), represents an alternative to multiple inheritance for handling variability. In our development process we use the related concept of partial class to implement the product lines. In languages such as C#, it is possible to split the definition of a class or interface over two or more source files (that can be in different packages). Each source file contains a section of the class definition, and all parts are combined when the application is compiled to form the final class. Using C#, if the project that implements the product line is organized in packages with partial classes (common and variants), for deriving a given specific product, it is enough to reference the packages that correspond directly with the selected configuration of the feature model. This technique connects feature models with the final code through the intermediate design models, thus solving the additional requirement of Subsection 2.1.

Though the use of these techniques is enough to organize the common and variable aspects of the SPL design in packages, the traceability management requires careful attention. For this reason, an automated approach to obtain the traceability links complete our approach and is presented in Subsection 2.3.

2.3 Automatic Generation of the SPL Package Architecture

To facilitate the work of the software engineers, it is possible to transform the feature models (and sub-models) into the SPL package architecture using a model driven based strategy, thus obtaining as a result the traceability links between the variable features and the design packages.

The definition of these transformations is based on OMG standard QVT [OMG, 05]. The target meta-model is UML 2. The feature meta-model proposed by [Czarnecki, 05b] has been selected as source, since it is powerful enough to represent the SPL variability and the simplicity of the related transformations. In this approach, the distinctive property of the relationships is the cardinality of the features and group of features, which distinguishes mandatory from variable features and guides the transformations. The features are characterized as Solitary or Grouped (following the mentioned meta-model). The SPL package architecture generation implies transforming:

- The Feature model into a UML model
- Each Root Feature into a Package
- Each variable Feature (Grouped or Solitary) into a package merged with its base package.

The strategy is based on the three subtypes of Feature defined in the cited meta-model. The root of every tree in a feature model (RootFeature) is transformed into a package. Then, a recursive transformation of solitary features and feature groups is carried out. Additionally, the presence of a group implies a new package per alternative *GroupedFeature*.

Figure 6 shows, graphically expressed, two of the QVT partial transformations. Initially, a feature model is transformed into a UML model and a general package that will contain all the variants. The transformation of each root feature into a base

package is the next step. An optional solitary feature is transformed into a new package merged with the previous package (the lower limit of *featureCardinality* is zero) (Figure 6a). If the minimal cardinality of the feature is not zero, the feature is always present and is not a variation point. Therefore, it is unnecessary to create a new package, and the related design elements will be included in the previous package. In both cases, the children (*Solitary* or *FeatureGroup*) features are recursively considered using the last package as reference (represented in the where clause). The rest of QVT definitions related to *FeatureGroup* (See Figure 6b) and *GroupedFeature* elements are similar. The key idea is that each variable feature (optional or alternative) requires an optional package in the SPL design. To generate the package configuration of each final product a similar strategy was used, starting this time from the corresponding feature sub model (i.e. the set of selected features that represent a concrete product of the SPL).

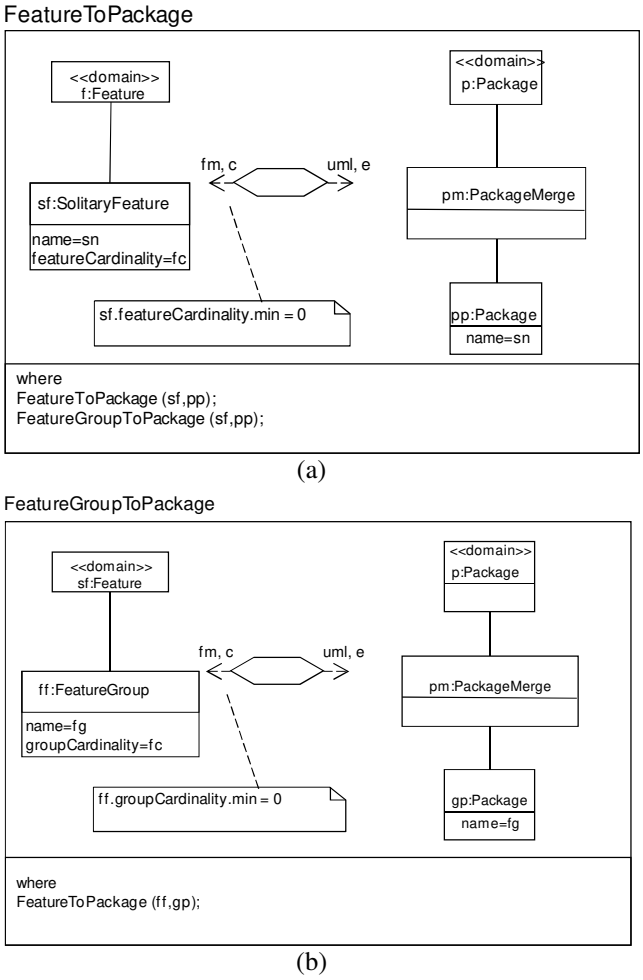

Figure 6: Graphical definition of two QVT transformations into UML packages

An implementation of these transformations was initially achieved starting from the XML files generated by the *fmp* eclipse plug-in [Antkiewicz, 04] using XSLT style sheets, and generating UML models in XMI format [Laguna, 07]. These models can be imported by UML CASE tools, yielding the SPL package architecture. These solutions, assuming a careful manipulation of the interchange files, permit the generation of the package configuration of final products following an independent language approach. However, a more practical solution is missed if we want to smoothly incorporate these techniques and transformations into the current software engineering tool box. The next Section presents a platform oriented implementation, integrated with MS Visual Studio, the reference tool for C# developers.

3 Feature Modeling Tool: Integration with Developer Tools

To make our proposal readily accessible to software engineers, we have developed a tool (and incorporated it into MS Visual Studio) that permits feature models to be created and the SPL package architecture and configurations to be generated, following the defined transformations. There are some open and commercial tools to manage and configure software product lines. The mentioned *fmp* plug-in is one of the best known. Commercial tools such as pure::variants[1] have a similar functionality for managing product lines. However, none of them includes the UML merge mechanism or integrates with existing C# tools. The aim was to manage a product line as a native project type provided by the development platform, including import/export capabilities, and facilities for model transformation and final product generation. Thus, this Feature Modeling Tool (FMT) makes the automation of many of the activities of the SPL development process possible (Figure 7).

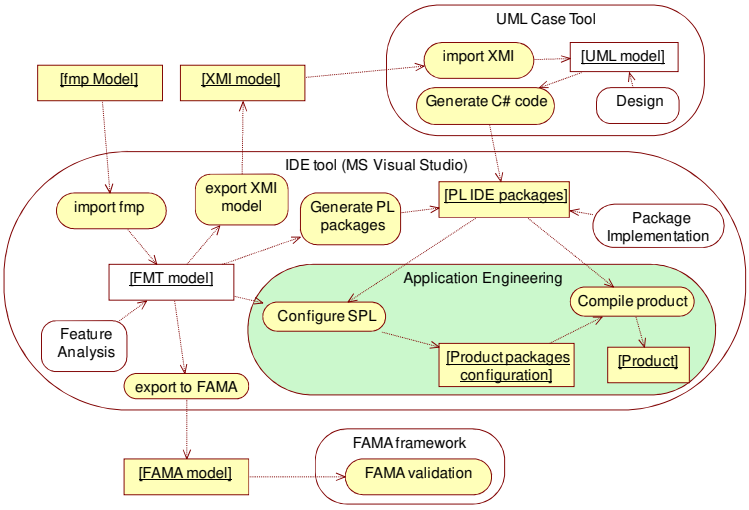

Figure 7: FMT functions as automated and manual activities in a generic SPL development process

[1] http://www.pure-systems.com/

Figure 7 depicts a global view of the FMT interfaces and functionalities. These functions support the activities of the *Product Line Engineering* and *Application Engineering* processes [Laguna, 03]. The manual intervention of the SPL expert engineer is needed to define the feature model (Feature Analysis activity) and the detailed design (Package implementation activity) of the SPL. The validation and package structure generation (in UML and IDE formats) are fully automated. Once the FMT and UML models of the SPL are defined, and the package implementation details completed, the application engineer can configure (selecting the desired featured) and compile each concrete product. This sub-process is fully automatic thanks to the integration of FMT in Visual Studio as shown in Figure 7 (the shadowed set of activities).

To implement FMT, we have used the Microsoft DSL tools [Cook, 07], part of the Microsoft Visual Studio SDK, intended for defining domain specific languages (DSL), and for creating the associated editors and utilities for these languages. The interface of FMT is mainly visual (Figure 8) but incorporates auxiliary tools, similar to the familiar *fmp* plug-in interface, and is compatible with it, allowing the direct import of *fmp* models.

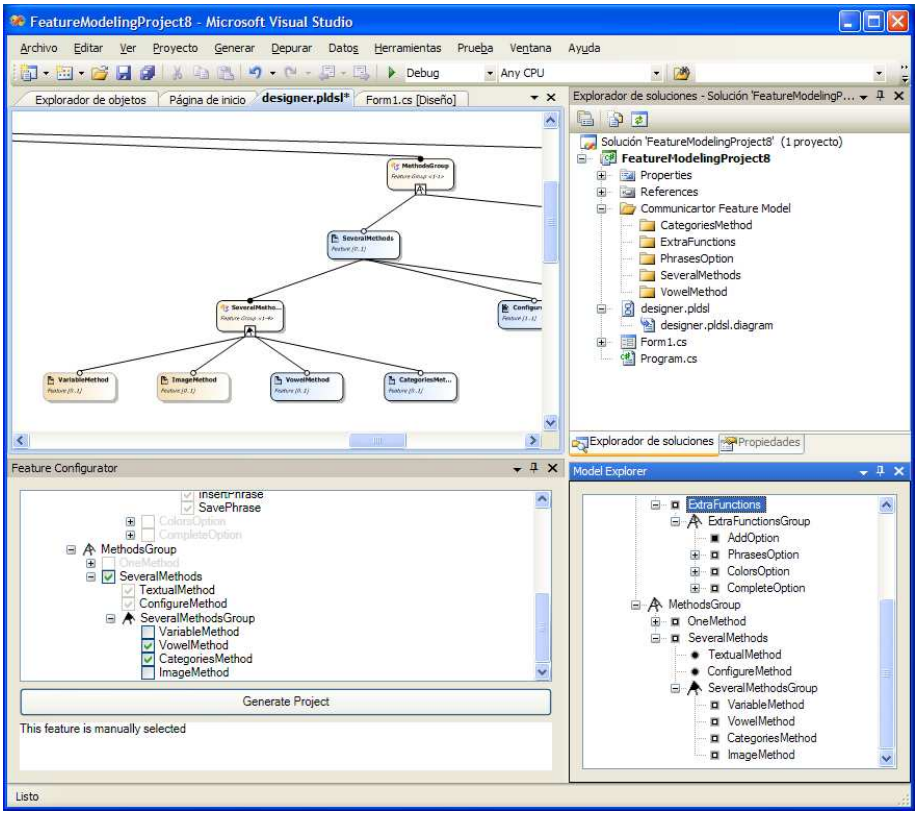

Figure 8: Feature Modeling Tool (graphical, native Solution Explorer, Configuration and Model Explorer views)

Figure 8 shows a general picture where the *Design* (graphical), *Solution Explorer* (the native Visual Studio package view), *Configuration*, and *Model Explorer* (the *fmp* style view) windows are shown. The advantages of FMT as compared with *fmp* are the direct integration in the Visual Studio IDE and the possibility of visual. The SPL package structure and the configuration files for the compiler can be automatically generated inside the conventional IDE. As additional benefits, import/export facilities allow communication with the FAMA validation framework [Trinidad, 08], export in XMI format to UML CASE tools, and import of *fmp* models. The tool and its documentation are available from the GIRO web[2].

4 Case Study: e-Commerce

In order to validate the proposed techniques and tool in a realistic situation, a case study has been carried out using a domain analysis described in [Lau, 06]. Using model superimposition, an SPL detailed design was proposed (concretely, class and activity diagrams were used), providing us with a very interesting starting point to contrast our techniques, since the packages that we must implement are imposed by an external independent study.

Our aim was not to implement dozens of packages, but to reach a result with enough variability to show that it is possible to develop a functional product line using a conventional development platform. At this moment, the common part of the product line and a dozen packages have been developed. Therefore, we can already generate hundreds of e-Commerce systems, from a minimal combination (the simplest purchase process) to a typical portal with registered users, shopping cart, credit card secure payment, multiple categories catalogs, search criteria, etc. The completed packages form a basic product family, though the variability can grow with the packages that are currently being developed. Figure 9 shows the developed features of the product line using the FMT Model Explorer view.

4.1 Product Line Design

The design adheres to the basic ideas of the design proposed by [Lau, 06], but organizing it in packages as generated by FMT. This has allowed us to focus our efforts on solving the practical problems of handling the web interface and the data access variability at the design and implementation levels. The SPL package architecture is shown in Figure 10. This structure is automatically generated (in XMI format) from the Feature Modeling Tool, using the incorporated transformations. The number of included features (near 40) is noticeably greater than the number of generated packages, as the design elements linked with mandatory features are included in pre-existing packages. We have completed the development of the common and optional packages of Figure 10.

Most packages are optional and mutually independent. However, more complex situations have also appeared: for instance, the *Electronic* and *Physical Product* packages correspond to an OR (1..2) structure in the feature model. This implies that

[2] http://giro.infor.uva.es/FeatureTool.html

at least one product type must be chosen. Another special situation is that the *Physical Product* package always requires the *Shipping Address* package to enable the effective shipment of the physical items. Though these restrictions are not shown in the package model, they are contemplated during the feature configuration process and automatically reflected (thanks to the traceability support) as valid or invalid feature (and thus package) combinations.

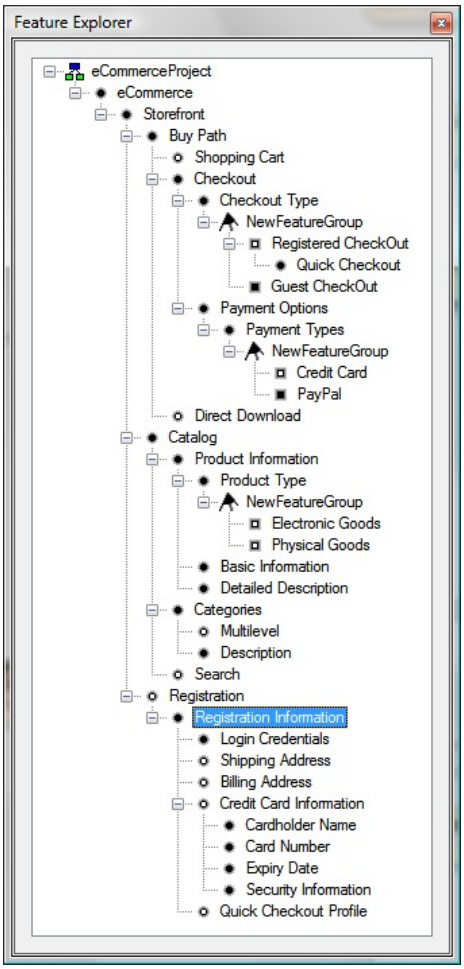

Figure 9: FMT Model Explorer view of the e-Commerce SPL case study

4.2 Product Line Implementation

The actual implementation of the e-commerce SPL has been done using the .NET/ ASP as platform, C# as language, and Microsoft Visual Studio as IDE tool. Variations in user interfaces cannot be implemented directly using packages and the partial class mechanism. To solve this problem, we have used a combination of templates, cascade style sheet files, and dynamic containers. In ASP.NET, it is possible to create master

pages that (combined with .css files) serve as templates to the web system. Each concrete product in the SPL will possibly have a different main page from the view point of the final users. The variability mechanism of the template is achieved by using dynamic containers (*ContentPlaceHolder*) that will be filled in a dynamic way as specified in the code of each concrete product.

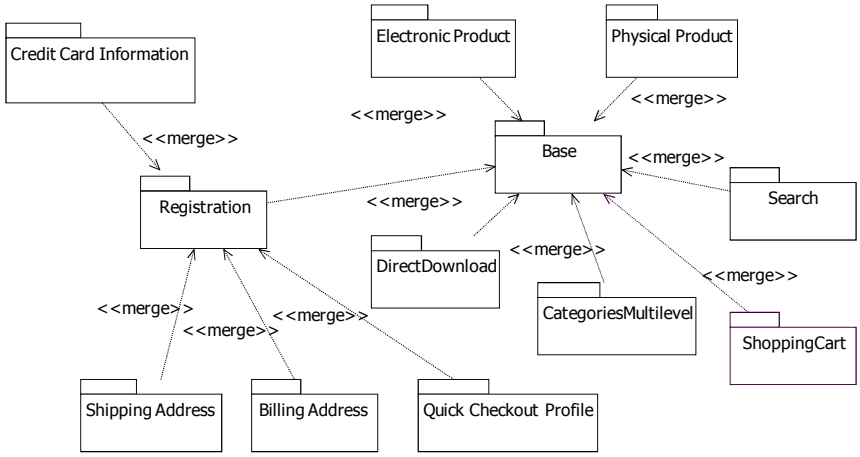

Figure 10: Packages of the electronic commerce product line

At the time of the configuration of each concrete product of the SPL, the compiler must recognize the necessary packages and the corresponding dynamic controls. To achieve this, Visual Studio uses a set of XML based configuration files which are used to indicate paths, packages, etc. In addition to these default configuration files, provided by the platform, other specific files have been added to each package. Thus, each page builds itself in a systematic way, using a name convention and the information about the necessary controls. The packages are organized and configured inside the IDE platform, using the FMT *configuration view* and this selection is automatically reflected through traceability links in the Visual Studio *solution explorer*. Once selected and validated, the compiler reads the project configuration files and generates the final specific product that can be deployed and installed in minutes in the production server. In Figure 11, two examples of final products with different degrees of complexity can be appreciated.

The results up till now include the generation of several hundred variants, by simply configuring and recompiling the generic SPL project into a concrete product. All the products include the basic purchase process. Also, several products have been instantiated and tested: Registered users, electronic or physical products, search facilities, credit card or *PayPal* secure payment methods, etc. The details of how to install and to configure a product can be consulted in the web of the GIRO group.

4.3 Lessons Learned

To summarize, a realistic e-commerce product line has been developed, using a seamless approach from the feature model to final implementation. At the same time,

the necessary implementation techniques to handle variability at code level have been established. The advantages provided by seamless development are complemented with the use of conventional tools (with the indispensable feature tool, also incorporated into the IDE). This combination of benefits can ease the adoption of the product line paradigm for the development of web systems by non specialist engineers and small companies in general.

The SPL development was planned by the authors and accomplished with the collaboration of graduate and last year students. The experience with these students has been satisfactory as they have reached the objectives with a reasonable effort (three to four months, four students working part time). Other similar web based product lines are in the development process. In particular, the feature model for the domain of associations has been established and the design completed (one student working six months). An unconnected approach is focused on mobile applications, where the interface and physical constraints impose new challenges to the SPL adoption (more effort in the interface variability design was necessary). In all the cases, the results are analogous and no insurmountable problems have been found by these novice developers. The experiences with these product lines were documented and made available in the GIRO web site.

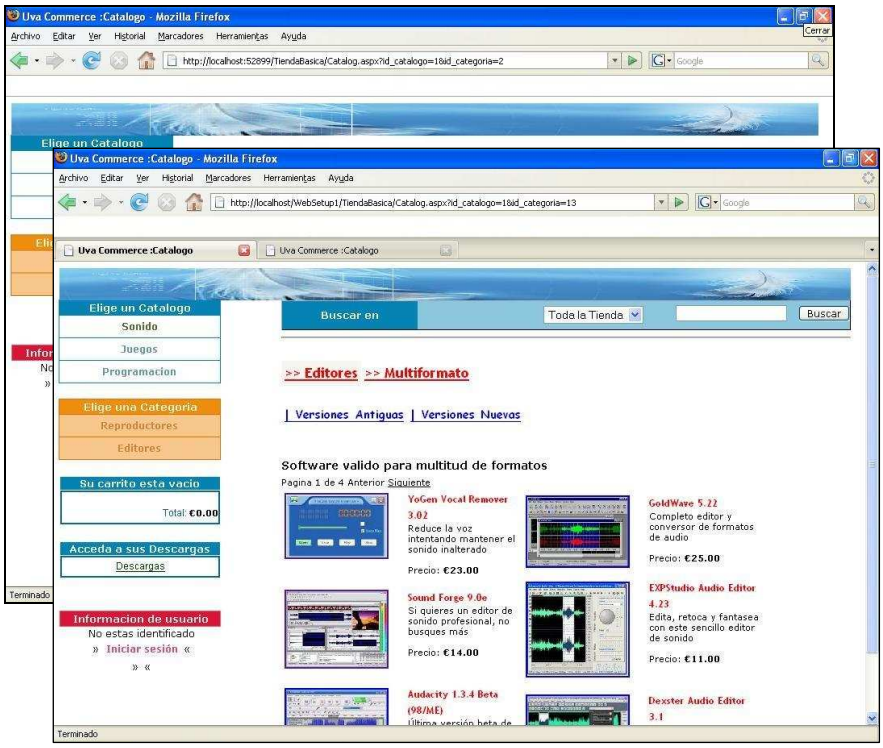

Figure 11: Two variants of the electronic commerce product line as seen in a web browser

Comparing these techniques with the original Lau work, our approach allows a better organization of the SPL design. In [Lau, 06] different details are mixed in a unique class diagram, forcing each fine detail to be marked with a stereotyped label, while we keep these details in separate packages, avoiding the need to label each element. When the product line grows, the number of labels increases exponentially (each variable feature forces the insertion of an undetermined number of stereotyped elements). Furthermore, alternative elements (several associations between two classes with different multiplicities) must be simultaneously present in the same diagram, obscuring the models and compelling the reinterpretation of the UML semantics. Using the package merge mechanism, each variable feature originates only a new package which contains all the details of each model increment. Each package is a valid UML model without inconsistencies. The complexity is delegated to the partial class mechanism and managed by the C# compiler.

5 Related Work

Though there are many projects that describe variability management mechanisms in terms of requirements and designs, few include implementation details. Different authors have proposed explicitly representing the variation points by adding annotations or changing the essence of UML. In the less intrusive approaches, either the mechanisms of UML are used directly (through the specialization relationship, the association multiplicity, etc.) or the models are explicitly annotated using stereotypes. For example, [John, 02] suggested the application of use case templates to represent the variability in product lines using stereotypes, although the authors do not distinguish between optional, alternative or obligatory variants. Concerning structural models, [Clauß, 01] proposes a set of stereotypes to express the variability in the design models: <<optional>>, <<variationPoint>> and <<variant>> stereotypes designate, respectively, optional, variation points (and their sub-classes), and variant classes. Gomaa makes extensive use of this approach, extending the use of stereotypes to all the UML models. He uses the stereotypes <<kernel>>, <<optional>> and <<variant>> (corresponding to obligatory, optional, and variant classes) [Gomaa, 00], [Gomaa, 04]. The advantage of these approaches is that conventional UML tools can be used to describe the SPL models, permitting the variability to be traced at the different levels, though the traces must be manually maintained. However, on many occasions, an optional feature can require a combination of attributes and/or methods in an existing kernel class. Neither do they solve either the requirement of a one-to-one correspondence between the different models.

Alternatively, other authors modify the original UML meta-model to adapt it to the SPL requirements. [Von der Maßen, 02] proposed using new relationships ("option" and "alternative") and the consequent extension of the UML meta-model. On the other hand, [Halman, 03] defends the modification of use case models to orthogonally represent the variation points (using a triangle symbol with different annotations). Another solution, proposed by [Czarnecki, 05a], consists of annotating the UML models with presence conditions, so that each variable feature is reflected in one or, more realistically, several parts of a diagram (perhaps a class, an association, an attribute, etc. or a combination of elements). This technique does not artificially

limit the representation of a variant with a unique element and even the color code helps to emphasize the implications of choosing a certain option. However, this visual help is not scalable when more than a dozen variants are handled. On the other hand, it is necessary to introduce auxiliary elements in the UML meta-model and CASE tools.

With a similar intention, FeatureMapper [Heidenreich, 08] is a tool that uses a "recording" mechanism to manually register the links between features and portions of the design UML diagrams. In these approaches, no automatic transformation facilities are provided.

Recent work in the AMPLE project proposes a variability modeling language (VML) inspired by the mentioned work of Halman, which supports first-class representation of different forms of design variability [Loughran, 08]. The language provides mechanisms to reference variation points in multiple design views and to define compositions, using fine-grained primitives such as *connect, add, remove*, etc and coarse-grained primitives, such as *package merge.* This last primitive is similar to our original proposal [Laguna, 07] but uses a package per features (the authors do not differentiate common from variable features, creating models that are necessarily bigger). VML include *require* and *exclude* dependencies that can be managed directly by the feature models. The result is a new domain specific language that manipulates UML elements, but using them from outside the standard UML.

A completely different approach, focused on implementation instead of requirements or design, is the Feature Oriented Programming (FOP) paradigm [Batory, 04]. The variable features are implemented as increments (refinements) in a java-like language. Starting from a base class, these increments are combined using a set of tools, provided with the AHEAD[3] tool suite. Other commercial tools, such as Big-Lever Gears or pure::variants offer similar functionalities. These approaches have in common the absence of high level models that show the SPL global organization.

As a general observation, though many of these solutions are valid, the learning of new modeling or implementation techniques and the need for specialized CASE and IDE tools represent barriers for the adoption of the SPL approach in many organizations.

Concerning the tool support aspect, there are several UML modelling tools that include merge facilities. Although their original intention is to help analysts to solve conflicts between two versions of the same model, these tools can be used to trivially merge SPL packages (that cannot have conflicts between compatible packages if the SPL is well designed). IBM Rational Software Architect[4] uses the concept of model fragment. The Magicdraw merge plug-in[5] and Enterprise Architect[6] (that supports integration with a version control system such as CVS operating with XMI files) offer a similar functionality. In all cases, the fragments are necessarily stored in physically separate files and must be merged using a wizard or plug-in. This constraint is logical for the original intention of the merge facilities but complicates the management of the SPL. Naturally, no support for feature models and its transformation into package

[3] http://www.cs.utexas.edu/users/schwartz/ATS.html
[4] http://www-01.ibm.com/software/rational/
[5] http://www.magicdraw.com/merge
[6] http://www.sparxsystems.com

models are provided. FMT include the merge facility into the development IDE, managing all the packages in a unique SPL model and making the merge details transparent for the application engineer during the configuration process.

6 Conclusions

In this work, the viability of package merge and partial class mechanisms as variability support in product line development has been shown. The use of the proposed mechanisms, combined with model driven transformations, enable the automated generation of each product from the features configuration and the structure of the product line itself. Furthermore, the use of conventional CASE and IDE tools can simplify the adoption of this paradigm, avoiding the necessity of specific tools and techniques as in the alternatives mentioned in Section 5.

As a part of this work, a Feature Modeling Tool has been developed and incorporated into the Visual Studio IDE. This direct integration allows the UML package model structure to be automatically generated and the final products to be configured from the Feature Modeling Tool. Therefore, the configuration process is more transparent and straightforward for the application engineers.

The approach has been successfully applied to the design and implementation of a product line in the web applications domain. The case study was based on a previous feature analysis on e-commerce domain, published in the literature, avoiding any possible bias in the feature model. The main goal was achieved: to develop a product line from beginning to end, defining the specific domain techniques to handle the variability at the implementation level. Finally, the presence of a plug-in that can be directly used by the application engineer from its conventional developing IDE, avoids the need of specific formation in new tools for these software engineers as the experience with recent graduates has shown.

7 Future Work

Current work includes the completion of other product lines with industrial or social interest, and the enrichment of the e-commerce case study. In this case, the objective is to evaluate the scalability of the proposal as the variable features increase (which implies an exponential increase in the number of final products). FMT is being used in these projects and some enhancements are being tested. In particular, the automatic generation of user interfaces by combining XML partial files (based on the XForms W3C standard[7]) and translating them into the HTML specific version of the interface for each final product.

The implementation of this approach with other programming languages different from C# is a pending work, to allow UML (language neutral) modeling/design tools to be connected with alternative IDEs. The Java language and the AHEAD suite are being adapted to our needs in alternative SPL projects with promising results, as the structure organization of Java directories can be seen as an implementation of the SPL package architecture (though AHEAD does not use that concept), and the suite

[7] http://www.w3.org/MarkUp/Forms/

utilities permit a final Java class to be obtained from several *jakarta* files (an extended version of Java), applying the AHEAD composition tools.

Acknowledgements

This work has been funded by the Spanish *Ministerio de Ciencia e Innovación* (TIN2008-05675).

References

[Antkiewicz, 04] Antkiewicz, M., Czarnecki, K.: Feature modeling plugin for Eclipse, In OOPSLA'04 Eclipse technology exchange workshop (2004).

[Batory, 04] Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling Step-Wise Refinement, IEEE TSE, June 2004.

[Bosch, 00] Bosch, J.: Design & Use of Software Architectures. Adopting and Evolving a Product-Line Approach. Addison-Wesley. 2000.

[Chung, 00] Chung, L., Nixon, B., Yu, E., Mylopoulos, J.: Non-Functional Requirements in Software Engineering, Kluwer Academic Publishers, 2000.

[Clauß, 01] Clauß, M.: Generic modeling using Uml extensions for variability,. In Workshop on Domain Specific Visual Languages at OOPSLA, 2001.

[Clements, 01] Clements, Paul C., Northrop, L.: Software Product Lines: Practices and Patterns, SEI Series in Software Engineering, Addison-Wesley, 2001.

[Cook, 07] Cook, S., Jones, G., Kent, S., Wills, A. C.: Domain-Specific Development with Visual Studio DSL Tools (Microsoft .NET Development Series), Addison-Wesley Professional, 2007.

[Czarnecki, 05a] Czarnecki, K., Antkiewicz, M.: Mapping Features to models: a template approach based on superimposed variants, In proc. of GPCE'05, LNCS 3676, Springer, 2005, pp. 422-437.

[Czarnecki, 05b] Czarnecki, K., Helsen, S., Eisenecker, U.: Formalizing cardinality-based feature models and their specialization, In Software Process Improvement and Practice, Vol. 10, No. 1, 2005, pp.7-29.

[Gomaa, 00] Gomaa, H.: Object Oriented Analysis and Modeling for Families of Systems with UML. InW. B. Frakes, editor, IEEE International Conference for Software Reuse (ICSR6), June 2000, pages 89–99.

[Gomaa, 04] Gomaa, H.: Designing Software Product Lines with UML, Addison Wesley, 2004.

[Halmans, 03] Halmans, G., Pohl, K.: Communicating the Variability of a Software-Product Family to Customers, Journal of Software and Systems Modeling 2:1, 2003, pp. 15-36.

[Heidenreich, 08] Heidenreich, F., Kopcsek, J., Wende, C.: FeatureMapper: Mapping Features to Models, In Companion Proceedings of the 30th International Conference on Software Engineering (ICSE'08), Leipzig, Germany, May 2008.

[Hofmeister, 99] Hofmeister, C., Nord, R.L., Soni, D.: Describing Software Architecture with UML. WICSA 1999, pages 145-160.

[Jacobson, 97] Jacobson I., Griss M. and Jonsson P.: Software Reuse. Architecture, Process and Organization for Business Success. ACM Press. Addison Wesley Longman. 1997.

[John, 02] John, I., Muthig, D.: Tailoring Use Cases for Product Line Modeling, Proceedings of the International Workshop on Requirements Engineering for Product Lines 2002 (REPL'02). Technical Report: ALR-2002-033, AVAYA labs, 2002.

[Kang, 98] Kang, K. C., Kim, S., Lee, J. y Kim, K.: FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures, Annals of Software Engineering, 1998, 5:143-168.

[Kang, 90] Kang, K. C., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study, Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute (Carnegie Mellon), Pittsburgh, PA 15213, 1990.

[Laguna, 03] Laguna, M.A., González, B., López, O., García, F. J.: Introducing Systematic Reuse in Mainstream Software Process, IEEE Proceedings of EUROMICRO, 2003, pp: 351-358.

[Laguna, 07] Laguna, M.A., González-Baixauli, B., Marqués, J.M.: Seamless Development of Software Product Lines: Feature Models to UML Traceability, GPCE 07. Salzburg, Austria, 2007

[Lau, 06] Lau, S.: Domain Analysis of E-Commerce Systems Using Feature-Based Model Templates, MSc Thesis, ECE Department, University of Waterloo, Canada, 2006.

[Loughran, 08] Loughran, N., Sánchez, P., Garcia, A., Fuentes, L.: Language Support for Managing Variability in Architectural Models, in Software Composition, Springer LNCS 4954, pp. 36--51, 2008.

[OMG, 03] Object Management Group, OMG: Unified modeling language specification version 2.0: Infrastructure, Technical Report ptc/03-09-15, OMG, 2003.

[OMG, 05] Object Management Group and QVT-Merge Group: Revised submission for MOF 2.0 Query/View/Transformation version 2.0, Object Management Group doc. ad/2005-03-02, 2005.

[Sochos, 04] Sochos, P., Philippow, I., Riebish, M.: Feature-oriented development of software product lines: mapping feature models to the architecture, Springer, LNCS 3263, pp. 138-152, 2004.

[Trinidad, 08] Trinidad, P., Benavides, D., Ruiz Cortés, A., Segura, S., Jimenez, A., FAMA Framework, SPLC, 2008, pp. 359.

[von der Maßen, 02] von der Maßen, T., Lichter, H.: Modeling Variability by UML Use Case Diagrams, Proceedings of the International Workshop on Requirements Engineering for Product Lines 2002 (REPL'02). Technical Report: ALR-2002-033, AVAYA labs, 2002.