Feature-Oriented Development of Software Product Lines: Mapping Feature Models to the Architecture

Periklis Sochos, Ilka Philippow, and Matthias Riebisch

Technical University Ilmenau, Process Informatics, Postfach 10 00 565
98684 Ilmenau, Germany
{Periklis.Sochos, Ilka.Philippow, Matthias.Riebisch}@tu-ilmenau.de
http://www.theoinf.tu-ilmenau.de/∼pld

Abstract. Software product lines (PLs) present a solid approach in large scale reuse. Due to the PLs' inherit complexity, many PL methods use the notion of "features" to support requirements analysis and domain modelling (e.g. FODA, FORM, FeatuRSEB). Nevertheless, the link between features and architecture remains weak in all methodologies, with a large impact on the traceability of high-level concerns in respect to lower-lever architectural structures. This paper provides an analysis on the state of the art of feature-oriented PL methodologies from the point of view of the linkage between feature models and architecture. Based on the identified shortcomings it introduces an approach to allow a strong mapping between features and architecture. The approach makes use of extensions in the feature modelling techniques and adopts plug-in architectures as a means of mapping feature structures and at the same time satisfying the demanded PL variability and flexibility.

Keywords: Software product lines, product line methods, feature modelling, separation of concerns, feature-architecture mapping, generative programming, plug-in architectures.

1 Introduction

Software product lines (PLs) replace the various separately-developed systems of a domain. Thus PLs embed at least the additive complexity present in each of these systems, posing a challenge in their development and demanding extensive variability. Additionally, PLs must follow all principles of modern software, being flexible, extendible and maintainable. In order to keep a balance between these requirements and virtues, PLs must adhere to a high level of abstraction: alone the variability and size of PLs impose the use of explicit domain modelling techniques and the development of a solid architecture. Many PL development methodologies have established "features" and "feature modelling" for this purpose e.g. FODA, FORM, FeatuRSEB.

In the current PL methods, the high level of abstraction obtained by feature domain analysis is constrained to the simplification and representation of the PL requirements or as a vague guide for design.

This paper aims to clarify the state of the art feature-oriented PL methods' insufficiencies in respect to the linkage between feature models and architecture and to present an approach to "bridge" the gap between these two artifacts. The proposed methodology involves the adoption of a modular plug-in architectural structure in synergy with feature modelling techniques. The method bears the name "Feature-Architecture Mapping" (FArM-reads farm).

Section 2 provides a rational for the selection of features as the main abstraction construct and the importance of a *strong* mapping between feature models and architecture. Section 3 illustrates the development of the main feature-oriented PL methods up to this point. Sections 4 until 7 analyze the previously identified methodologies from the point of view of the linkage between feature model and architecture. In section 8 an overview of the development process of FArM is presented. Finally, sections 9 and 10 provide the conclusions and further work.

2 Feature Models & Architecture in Software Product Lines

Software product lines centralize upon the idea of designing and implementing a family of systems to produce qualitative applications in a domain, promote large scale reuse and reduce development costs.

2.1 The Role of Features

Achieving these goals in the space of a domain makes the need of preserving a high level of abstraction even more evident. Based on this rational, the concept of a *feature* is introduced.

A formalized definition of a feature in the software field is given in [7]: "*a logical unit of behavior that is specified by a set of functional and quality requirements*". Adding to this, comes the definition of features from [18]: "*A feature represents an aspect valuable to the customer...*".

Furthermore, features may be structured in *feature models*. These are hierarchical tree illustrations of features. There exist many variants of feature modelling conventions (see [7], [11], [21], [22], [24]). A sample feature model of an IDE (Integrated Development Environment) PL showing the previously mentioned conventions, with the *multiplicity* extension (see [18] and [19]), is given in figure 1.

Summarizing, features may serve as a means of:

- modelling large domains
- managing the variability of PL products
- encapsulating system requirements
- guiding the PL development
- driving marketing decisions
- future planning
- communication between system stakeholders

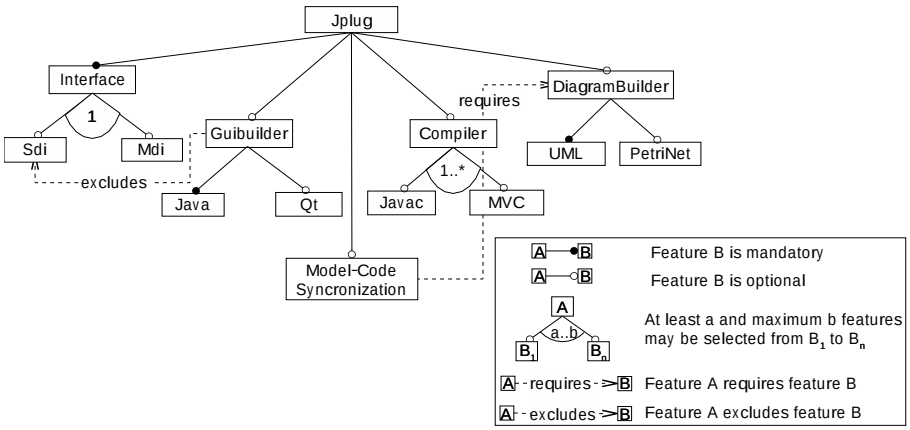

Fig. 1. A sample feature model

2.2 The Role of Feature Model & Architecture Mapping

During the design phase of a PL, one needs to identify all important aspects of the system and based on the imposed quality requirements, to device the main architectural abstractions and the connections between them.

In order to achieve a long-life, maintainable system, one must adhere to the rule of *separation of concerns* i.e. achieve such an architectural structure, where ideally each architectural component encapsulates exactly one *concern*. In PLs, that would ideally be one *feature*[1].

Nowadays, software technologies (e.g. Object-Orientation) make it extremely difficult to achieve a pure one to one relation between feature models and architectures [28]. Therefore, it is vital for the PL system to achieve, if not a pure one to one relation, at least a strong mapping between features and architectural components.

3 Feature-Oriented Product Line Methods

With the term *Feature-Oriented* product lines we stress the fact that this paper is concerned with the PL methods making intensive use of the notion of features and feature modelling. Looking at the evolution of PL methods in relation to each other, one could extract the picture shown in figure 2.

Rather than providing a complete reference to all possible variations of the feature-oriented methodologies, this paper concentrates on those methods using features as a main artifact in their processes, having a wide industrial acceptance and providing sufficient documentation.

[1] Experience from real life [14] shows that high maintainability may be achieved, when at least a one-to-many relation between features and architectural components is present.

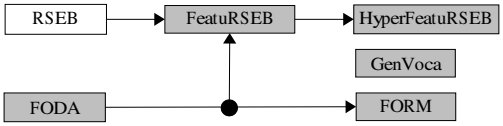

Fig. 2. Development of Feature-Oriented Product Line Methods

The identified methods are FODA (Feature-Oriented Domain Analysis), FeatuRSEB (Featured RSEB), HyperFeatuRSEB (Hyper Featured RSEB), GenVoca and FORM (Feature-Oriented Reuse Method). Note that the RSEB (Reuse-Driven Software-Engineering Business) [15] method is provided only for completeness. It is based mainly on use-cases and therefore it is not explicitly analyzed in this paper.

4 FODA and FORM

FODA is a domain analysis method. It focuses on providing a complete description of the domain features, paying less attention on the phases of design and implementation [1]. The feature modelling notation used in FODA does not include the representation of basic processes in a system's architecture, e.g. interactions between architectural components implementing features. Furthermore, FODA lacks a concrete description of the transition from a feature model to an architecture.

FORM comes as a concretization of the FODA processes. It provides guidelines for the creation of the feature model, design and implementation phases. In [27] FORM's authors provide a description of these phases. As presented in this source, FORM performs an analysis of a domain's features and attempts to provide a mapping between features and architectural components: "*By designing each selectable feature as a separate component, applications can be derived easily from product-line software. If there is difficulty in establishing this relation, the feature must be refined into specific features so that features can be easily mapped into architectural components.*"

FORM provides no concrete description of the above mentioned process. Furthermore, feature interactions are superficially addressed by FORM. No explicit support is provided by the method for the construction of an architecture to conform to the structure of a feature model. FORM's main focus, as given in [23], is not providing a clear mapping between the feature model and the architecture, rather concretizing the FODA processes of design and analysis from a marketing perspective.

FODA and FORM remain vague on the matter of mapping feature models to architectural elements. FODA concentrates on the modelling of the domain concepts and FORM does not place enough focus on describing such a process.

5 FeatuRSEB

FeatuRSEB merges the RSEB and the FODA methods. Its main goal is to provide a central model for the synchronization of the RSEB processes and at the same time model the commonality and variability within the RSEB. This is achieved through the introduction of the feature model.

FeatuRSEB is divided, as most PL methods, into two main processes, namely, Product Line Engineering and Product Engineering, where the former develops the PL assets and the latter makes use of them to build PL products.

The Product Line Engineering phase is initiated by storing all requirements of the product line in a common repository in the form of use-cases. The PL requirements are also ordered in features on a feature model. Features and use-cases are linked with traceability links. The PL architecture is derived from the use-cases following the "Layers" architectural pattern [8], where each layer consists of components containing object-oriented classes. Traceability links are then assigned between the derived classes and their use-cases.

Thus FeatuRSEB performs the mapping between feature models and architectural elements exclusively through traceability links pointing to use-cases, which in turn point to classes within the architectural elements.

This way of mapping a feature model to the architecture, although it does present a step towards the right direction, has a number of disadvantages. Namely, the number of traceability links in a PL very soon becomes extremely large. The creation, management and maintenance of the traceability links, even for a normal sized PL, is an overwhelming task and its resolution is not addressed by FeatuRSEB.

6 HyperFeatuRSEB

HyperFeatuRSEB combines the Hyperspace and FeatuRSEB methods. It utilizes Hyperspace techniques to map features to architectural component. Because of this fact, this section will explore HyperFeatuRSEB in more detail. Before considering specific aspects of HyperFeatuRSEB, section 6.1 will provide the needed terminology for the Hyperspace Approach.

6.1 The Hyperspace Approach

The Hyperspace Approach [28] was developed by IBM to achieve a *multiple-separation of concerns*. A concern can be anything that is of importance to the system stakeholders, from requirements to features or implementation details.

The Hyperspace approach allows the definition of a multi-dimensional *hyperspace*, where all concerns are included. A *hyperslice* may encapsulate one of these concerns, and the combination of many hyperslices through *integration relationships* yields a product containing all desired concerns, defined as a *hypermodule*.

These concepts can be easier understood by means of an example throughout the description of the HyperFeatuRSEB method.

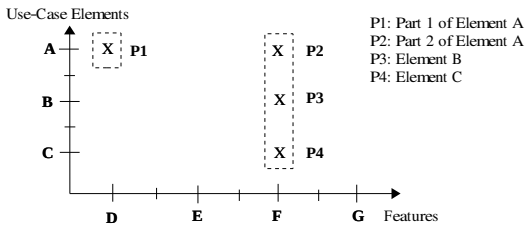

Fig. 3. A two-dimensional hyperspace. A hyperslice is marked with a dashed line and encapsulates exactly one feature.

6.2 The HyperFeatuRSEB Method

HyperFeatuRSEB [5] identifies the problem of separation of concerns in FeatuRSEB and integrates FeatuRSEB with the Hyperspace approach after providing a UML extension of this method, the HyperUML [6].

HyperFeatuRSEB's structure is similar to that of FeatuRSEB. This time FeatuRSEB's use-cases are broken down to actors, use-cases and activity diagrams. Each of these use-case elements are assigned to their belonging features and are ordered in a two-dimensional hyperspace, as shown in figure 3.

The product line engineering derives the PL's common architecture from the complete use-case model. Separate processes, coordinated by the PL engineers, derive the object model for each hyperslice (i.e. feature) from the partial use-cases within each hyperslice.

Thus, we have one hyperslice (i.e. feature) containing the object model elements belonging to it, which assures a one to one relation between feature model and architectural elements. The inadequacies of this method are illustrated through an example.

Figure 4 shows a partial feature model of a PL made with HyperFeatuRSEB and the respective hyperslices encapsulating the parts of the architectural elements for each feature.

As shown in figure 4, if a customer chooses to have an IDE (Integrated Development Environment) with an Sdi (single document interface), the "empty" method `GetSelectedText()` in the `Core` hyperslice, marked by the Hyperspace construct `Unimplemented`, will be replaced by the "implemented" `GetSelectedText()` method in the `Sdi` hyperslice. When the customer selects the Mdi (Multi document interface) feature, the respective substitution will take place.

As a result, the end product will contain either the implementation of the Sdi or the Mdi feature.

6.3 HyperFeatuRSEB and the Hyperspace Approach Open Issues

The shortcomings of the methods become evident when a "non-additive" change occurs. With the term non-additive, we mean, for example, a change that causes a method to be removed and replaced by others. Such a change can be seen in figure 5.

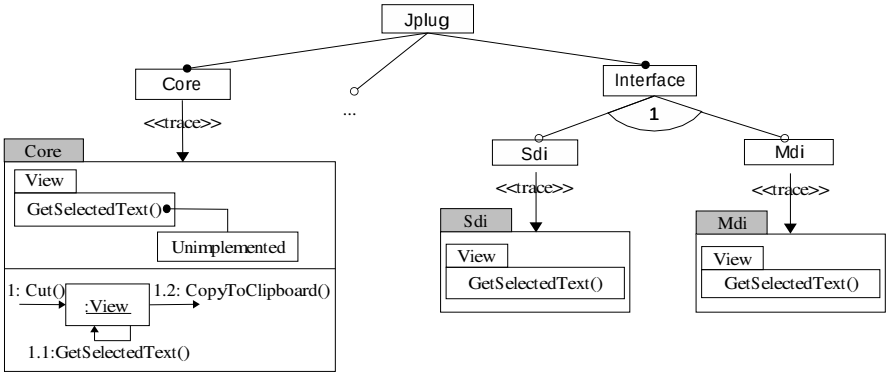

Fig. 4. A partial feature model for an IDE product line, along with the features' hyperslices from an architectural perspective.

The implementation of the `GetSelectedText()` method of the Mdi feature in figure 4 needs to be removed and replaced by the `GetActiveFrame()` and `FindMarkedText()` methods, as shown in figure 5. A hyperslice is "encapsulated", so a change of its internal structure is perfectly legal.

If a customer now chooses the Mdi feature, the `View` class in the `Core` and `Mdi` hyperslices can not be *merged* anymore. The `GetSelectedText()` method in the `Core` has to be changed in accordance to the Mdi's feature methods to `GetActiveFrame()` and `FindMarkedText()`. But now merging the `Core` and `Sdi` features is not possible anymore. This means that the Sdi feature must also be changed and surprisingly in an absurd way.

This example has led us to the following conclusions:

Maintainability In the work on HyperFeatuRSEB [5] a software product line has been implemented to illustrate the strengths and weaknesses of the method and serve as a means of comparison to the FeatuRSEB method.

An analysis on the source code of this system has shown that the PL has 1243 such unimplemented methods from overall 4197, shared between various

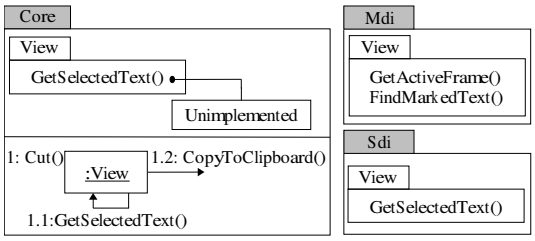

Fig. 5. A non-additive change within a hyperslice causes cascading changes in other hyperslices (i.e. features).

combinable features, in other words: "*1 out of 3 method changes causes at least 2, at most 19, on average 4 features to change*".

The way of the Hyperspace approach to create a separation of concerns has led to an extreme sensitization of the underlying system's architecture to non-additive changes. This points out the poor maintainability of the produced product lines.

Feature Interaction Hyperslices must have *declarative completeness* [28]: a hyperslice *must* contain all structure (e.g classes, methods, etc) definitions or partial implementations of all other hyperslices to which they refer to. This fact enforces the inclusion of identical, `unimplemented` methods, belonging to other hyperslices. Therefore, hyperslices are not encapsulated from their environment, on the contrary, they illustrate high coupling, since small changes in a hyperslice cause changes in other hyperslices. These interactions are neither explicitly modelled in HyperFeatuRSEB, nor are they taken into consideration in the system's development.

Tool Support The tool support needed for the creation and management of hyperspaces, as well as for the definition, development, maintenance and management of hyperslices is not provided at the moment. The HyperJ tool accessible from the IBM web site [30] and described in [29] is still in an immature phase and not in the position to support a PL development process. There is clear need for professional case tool support for the method's implementation.

7 Aspect-Oriented Programming and GenVoca

Other generative programming techniques could also be used to provide a separation of concerns. This section examines the Aspect-Oriented Programming and the GenVoca approaches.

7.1 Aspect-Oriented Programming

One could also achieve a separation of concerns using the Aspect-Oriented Programming technique [25] and effectively perform an integration with one of the aforementioned PL methodologies, as in HyperFeatuRSEB.

Aspect-Oriented Programming makes use of *aspects*. These are code descriptions of a concern, which are weaved into the system and respectively affect its behavior. There exists a special grammatical syntax [26], as well as a possible UML extension [20] for the modelling of an aspect-oriented implementation.

One of the issues with this approach is that the aspect-modules are difficult to understand and maintain. The developers need to maintain the system's code and also the extra developed aspects, "inflating" the system's complexity.

7.2 GenVoca

The GenVoca [2] method is based on similar principles as the Aspect-Oriented Programming and the Hyperspace approach. Systems are built from layers containing programs [3]. Each program implements a feature. Combinations of layers allow the addition of the programs' functionality to compose a working system. The whole process is supported by the Jakarta tool suite [16].

The GenVoca modelling of PL systems is performed by means of a formal algebraic notation [3]. It is based mainly upon the same principles of class composition as the Hyperspace approach and therefore bares the same problems. Finally, the Jakarta tool suite proves to be in an immature state to support the actual development of a product line [5].

8 Feature-Architecture Mapping – FArM

The methodology presented in this section of the paper introduces solutions based on the identified deficiencies of the state of the art PL methodologies. It strives to achieve an efficient mapping between feature models and the architecture and at the same time preserve the qualities of a PL, like enhanced maintainability, ease of evolution and simplified product generation.

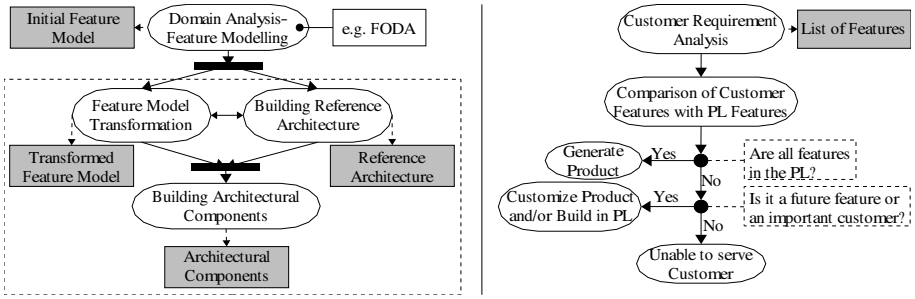

Fig. 6. The FArM Product Line and Product Engineering workflows.

8.1 Process Overview

FArM is divided into a Product Line Engineering and Product Engineering phases effectively managing the PL's development complexity. FArM's workflows for the Product Line and Product Engineering phases are illustrated in figure 6.

FArM may be applied right after the domain modelling stage of a PL's development cycle. The current state of the method supports the FODA domain analysis method [21].

After the development of an initial feature model, FArM proceeds with two concurrent bidirectional processes, namely the Feature Modelling Transformation and Building Reference Architecture processes. The result of these processes is a transformed feature model compliant with the newly built PL's reference architecture. The latter is a plug-in architecture.

At this point, each identified feature is implemented in exactly one architectural (plug-in) component in the Building Architectural Components process, following the guidelines defined in the transformed feature model and reference architecture artifacts.

In FArM's Product Engineering phase needs to be pointed out that if the customer's requirements are satisfied by the product line features, the product will be *generated* by simply plugging the feature-components to the plug-in platform.

8.2 Product Line Engineering

This section takes a closer look at the Product Line Engineering phase of FArM and its processes.

Feature Model Transformation The Feature Model Transformation process receives the initial feature model and performs a transformation based on predefined *transformation rules*. The goal of this process is to allow a logical ordering of the features and model the product line's feature interactions, as well as providing a complete list of possible non-costumer related features. The Feature Model Transformation is based upon the PL's requirements and high-level architectural decisions. This process runs concurrently with the Building Reference Architecture process, maintaining a bidirectional communication link among the two processes.

The transformation of the initial feature model is performed through the use of predefined transformation rules. These can lead to **adding** features, **integrating** features within other features, **dividing** features and **reordering** the hierarchy of features on the feature model.

FArM transformation rules are based on:

- **Grouping Features**: Grouping features must illustrate a logical relation to their sub-features (e.g. Interface–Mdi-Sdi in figure 1).
- **Quality Features**: Quality features must be integrated inside functional features (e.g. "performance" in an IDE could be integrated as a time limitation on the model-code synchronization feature and/or other features).
- **Architectural Requirements/Implementation Details**: The PL itself may impose the existence of a variety of architectural structures through requirements or implementation details. These should in turn yield features in the feature model to retain the one-to-one relationship between feature model and architecture (e.g. a "security" feature in the IDE PL to prevent unregistered use of the software, is mainly a concern of the developers. Thus it must be implemented is the PL architecture and reflected as a feature in the feature model, although it is not a customer visible feature.).

– **Interacts Relationships**: When referring to interacts relationships, we concentrate on the communication between features for the completion of a task. Transformations based upon such communication support the developers in the explicit modelling of the extra-hierarchical relationships between features and also transferring this knowledge to the system's architecture. Interacts relationships support also the design for maintenability as well as PL end-product instantiation.

The modelling of feature interactions is implemented in FArM through an extension to the feature model, namely the *interacts relationship*. An example of an interacts relationship is shown in figure 7. Information related to a FArM interacts relation can be of a textual or formal form (e.g. OCL) and must avoid going into implementation details. These are documented in the feature's respective architectural component's documentation.

All transformation rules defined within the Feature Model Transformation process strive to achieve a balance between the initial feature model and the changes needed to allow a strong mapping to the architecture, preserving the nature of the initial features.

Building Reference Architecture FArM's reference architecture is a plug-in architecture. The root feature of the feature model representing the product line is the architecture's plug-in platform. Each feature is implemented in exactly one plug-in component and adheres to the platform's plug-in format.

During this phase the developers have the task of defining the plug-in format and communication protocols for inter-component communication. Based on the requirements placed upon each feature and the needed interactions, each component is assigned and commits to the implementation of an interface to provide services to other components.

Messages sent to a parent component/feature are transmitted to the proper sub-component/feature, allowing the decoupling of the components and providing the needed flexibility for instantiating PL products.

The advantages of such a plug-in architecture, from the perspective of establishing a strong mapping between feature models and architectures are:

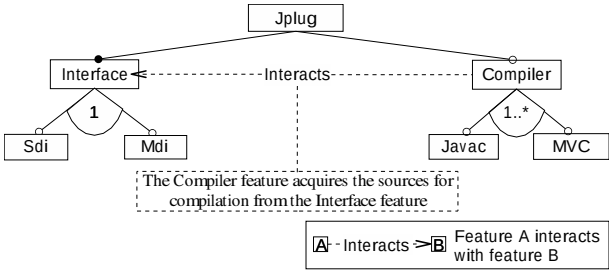

Fig. 7. An interaction relationship in a FArM transformed feature model.

– Allowing automatic product generation: Features plugged into the plug-in platform instantly compose PL products.
– Allowing the encapsulation of features in plug-in components.
– Decoupling of the features is achieved through the hierarchical plug-in structure.
– Component interfaces allow the modelling of feature interactions.

8.3 Feature-Architecture Mapping Solutions

Based on the description of FArM, this section will illustrate the solutions to the identified problems in the state of the art feature-oriented methodologies from the perspective of the mapping between feature models and architecture.

Feature Model-Architecture Transition FArM's main goal is to clarify the transition from a feature model to an architecture. This is achieved through a detailed description of the Feature Model Transformation and Building Reference Architecture processes, as well as the introduction of transformation rules and the support for the creation of the PL's architecture.

The state of the art methodologies analyzed in the previous sections provide insufficient description of the needed steps for this transition.

PL Maintainability HyperFeatuRSEB's maintainability issues are resolved within FArM through the use of a modular plug-in architecture.

Figure 8 illustrates the solution to the maintainability problem introduced in section 6.3. Now, each plug-in component representing a feature, defines an interface to allow the interaction between features. A change in the inner parts of the `Mdi` feature has no effect on the feature's interface.

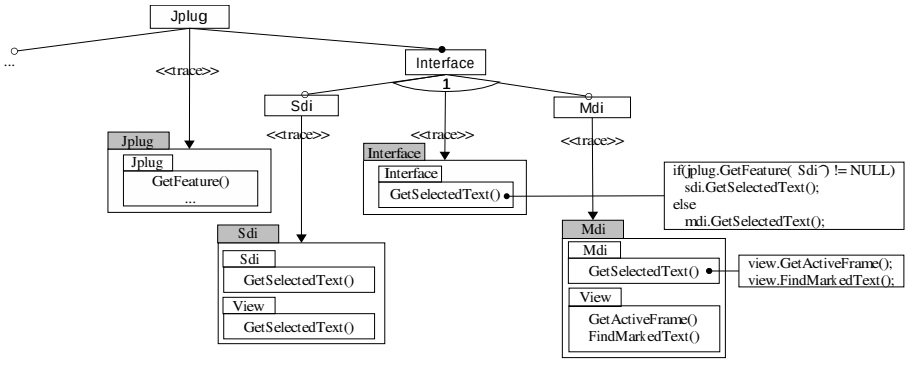

Fig. 8. A FArM solution to the maintainability issues of HyperFeatuRSEB.

Feature Interaction The issue of feature interactions is identified and accepted by FArM as a part of the PL system. It is therefore integrated within the development process in the Feature Model Transformation and Building Reference Architecture processes. Explicit modelling of the feature interactions is performed through the feature model extension *interacts* relationship and its reflection in the plug-in components' interfaces. Furthermore, this process can be supported from related work on resolving feature interactions [9], [31].

None of the state of the art methodologies directly addresses the issue of feature interactions. In FODA, FORM and FeatuRSEB this issue is not a separate part of the development process, while the Hyperspace approach in HyperFeatuRSEB claims to achieve a "clear" separation of concerns, neglecting feature interactions and causing the maintainability problems already discussed.

Tool Support FArM requires the use of a case tool for the development of the plug-in architecture and a feature modelling tool for construction and maintenance of the PL feature model. For both tools exist sufficient (semi-)commercial software solutions [17], [10], [12].

The issue of sufficient tool support was identified as one of the problems in the HyperFeatuRSEB and the GenVoca methods.

9 Conclusions

This paper has explored the main feature-oriented PL methodologies from the point of view of the linkage between feature models and architecture. Based on the identified shortcomings, an approach is introduced to allow a stronger mapping between a PL's feature model and architecture.

More precisely, although FODA and FORM provide a number of guidelines, their focus is not placed directly upon the resolution of a weak mapping between feature models and architecture. FeatuRSEB performs a mapping through the heavy use of traceability links, providing little support for the creation, management or maintenance of such constructs.

The integration of generative programming techniques in product line methodologies, like Aspect-Oriented Programming and GenVoca, proves to be insufficient for the development of a product line. Maintainability problems and complexity in the aspect-modules, as well as lack of modelling principles and sufficient tool support in the GenVoca environment, denote these issues.

HyperFeatuRSEB uses the Hyperspace approach to isolate features within hyperslices. The result of this process carries a number of disadvantages, e.g. low maintainability and immature tool support.

The Feature-Architecture Mapping method (FArM) introduced in this paper identifies the nature of the problems prevailing in the aforementioned methodologies and supports the mapping of features to the architecture with existing technologies.

Explicit feature modelling processes with well defined transformation rules support the smooth transition from a feature model to the architecture. The

interaction between features is documented on different levels: on the model level through the feature model extensions of the *interacts* relationships, and on an architectural level in the interfaces of the respective plug-in components.

Furthermore, support for the development of a flexible and maintainable product line architecture is provided through the proposed plug-in structure. Figure 8 shows the flexible nature of the FArM architecture: the `Interface` feature serves as a switch mechanism between the `Sdi` and `Mdi` features, thus enabling an automatic product instantiation. The method supports the generation of PL products, while the maintainability and the system's complexity remain manageable. Finally, the method allows the use of commercial tools.

10 Further Work

The next steps in the development of the Feature-Architecture Mapping (FArM) method include:

- The support of more domain analysis methods for the creation of the initial feature model.
- The formal definition of transformation rules for the transition from the feature model to the architecture.
- Integration of feature interaction resolution techniques in the FArM processes.
- The development of a process specifically for the support of the definition of component interfaces in respect to feature interactions.

A number of features from the IDE product line presented in the examples have already been implemented with FArM. An industrial case study is taking place at the point of this writing in the domain of mobile phones. More specifically, the plug-in platform of the Blackberry handheld is used for the development of client components with the FArM method. Publications will follow on this theme. Further work will also include the implementation of the method in various other domains, e.g. real-time system's, medical domain, etc. Finally, future work includes the precise definition of the method's limitations.

References

1. Atkinson, C: Component-based product line engineering with UML. Addison-Wesley (2002)
2. Batory, D. and Geraci, J. B.: Composition Validation and Subjectivity in GenVoca Generators. IEEE Transactions on Software Engineering (23)(2), 67–82 (1997).
3. Batory, D.; Lopez-Herrejon, E. R.; Martin J.: Generating Product-Lines of Product-Families, Automated Software Engineering Conference, Edinburgh, Scotland, 81–92 (2002)
4. Blackberry Handheld, http://www.blackberry.com/
5. Boellert, K.: Object-Oriented Development of Software Product Lines for the Serial Production of Software Systems (Objektorientierte Entwicklung von Software-Produktlinien zur Serienfertigung von Software-Systemen). PhD Thesis, TU-Ilmenau, Ilmenau Germany (2002)

6. Philippow, I.; Riebisch, M.; Boellert, K.: The Hyper/UML Approach for Feature Based Software Design. In: The 4th AOSD Modeling With UML Workshop. San Francisco, CA (2003)
7. Bosch, J.: Design & Use of Software Architectures - Adopting and Evolving a Product Line Approach. Addison-Wesley (2000)
8. Buschmann, F.: Pattern-Oriented Software Architecture: A System of Patterns. John Wiley & Sons (1996)
9. Calder, M.; Kolberg, M.; Magill, M.H.; Reiff-Marganiec, S.: Feature Interaction A Critical Review and Considered Forecast. Elsevier: Computer Networks, Volume 41/1 (2003) 115–141
10. Captain Feature, http://sourceforge.net/projects/captainfeature
11. Czarnecki, K.; Eisenecker, U.W.: Generative Programming. Addison-Wesley (2000)
12. DOME (DOmain Modelling Environment), http://www.htc.honeywell.com/dome/
13. Griss, D.; Allen, R. and d'Allesandro, M.: Integrating Feature Modelling with the RSEB. In: Proceedings of the 5th International Conference of Software Reuse (ICSR-5) (1998)
14. Pashov, I.: Feature Based Method for Supporting Architecture Refactoring and Maintenance of Long-Life Software Systems. Phd Thesis. Technical University of Ilmenau, Ilmenau, Germany, 2004 (submitted)
15. Jacobson, I.; Christerson, M.; Jonsson P.; and Oevergaard, G.: Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley (1992).
16. Jakarta Tool Suite. www.cs.utexas.edu/users/schwartz/
17. Rational Rapid Developer, http://www-306.ibm.com/software/awdtools/rapiddeveloper/
18. Riebisch, M.: Towards a More Precise Definition of Feature Models. In: Workshop at ECOOP. Books On Demand GmbH, Darmstadt, Germany (2003) 64–76
19. Streitferdt, D.; Riebisch, M.; Philippow, I.: Formal Details of Relations in Feature Models. In: Proceedings 10th IEEE Symposium and Workshops on Engineering of Computer-Based Systems (ECBS'03). IEEE Computer Society Press, Huntsville Alabama, USA (2003) 297–304
20. Suzuki, J. and Yamamoto, Y.: Extending UML with Aspects: Aspect Support in the Design Phase. In Proceedings of the Aspect-Oriented Programming Workshop at ECOOP '99 (1999).
21. Kang, K.; Cohen, s.; Hess, J.; Novak, W.; Peterson, A.: Feature-Oirented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh (1990)
22. Kang, KC; Kim, S.; Lee, J.; Kim, K.; Shin, E.; Huh, M: FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. Annals of Software Engineering, 5 (1998) 143–168
23. Kang, KC; Lee, J. and Donohoe, P.: Feature-Oriented Product Line Engineering. IEEE Software, Vol. 9, No. 4, Jul./Aug. (2002) 58–65
24. Kang, KC; Lee, K.; Lee, J.: FOPLE - Feature Oriented Product Line Software Engineering: Principles and Guidelines. Pohang University of Science and Technology (2002)
25. Kiczales, G.: Aspect-Oriented Programming. Springer-Verlag, In Proceedings of the 1997 European Conference on Object-Oriented Programming (ECOOP '97), (1997) 220–242
26. Kiczales, G.: Getting Started with AspectJ. Communications of the ACM (44)(10), (2001) 59–65

27. Kang, KC.; Lee, K.; Lee, J. and Kim, S.: Feature-Oriented Product Line Software Engineering: Principles and Guidelines. In: Domain Oriented Systems Development: Practices and Perspectives. Taylor & Francis (2003) 19–36
28. Ossher, H.; Tarr, P.: Multi-Dimensional Separation of Concerns and the Hyperspace Approach. In: Software Architectures and Component Technology. Kluwer Academic Publishers (2001)
29. Tarr P. and Ossher, H.: Hyper/J User and Installation Manual (2001)
30. Multi-Dimensional Separation of Concerns: Software Engineering using Hyperspaces. www.research.ibm.com/hyperspace/
31. Zave, P.: FAQ Sheet on Feature Interaction. AT&T (1999)