The Feature-Architecture Mapping (FArM) Method for Feature-Oriented Development of Software Product Lines

Periklis Sochos, Matthias Riebisch, Ilka Philippow
Technical University of Ilmenau
Software Systems/Process Informatics
Ilmenau 98684, Germany
{periklis.sochos, matthias.riebisch, ilka.philippow}@tu-ilmenau.de

Abstract

Software product lines (PLs) are large, complex systems, demanding high maintainability and enhanced flexibility. Nonetheless, in the state of the art PL methods, features are scattered and tangled throughout the system components, leading to poor maintainability. Additionally, the majority of PL methods support manual product composition, while the implementation of feature-level variability in PL products influences the system's conceptual integrity. Generative programming techniques do enhance flexibility, but on the cost of maintainability. The Feature-Architecture Mapping (FArM) method provides a stronger mapping between features and the architecture. It is based on a series of transformations on the initial PL feature model. During these transformations architectural components are derived, encapsulating the business logic of each transformed feature and having interfaces reflecting the feature interactions. The flexibility of FArM architectures is supported through the explicit integration of plug-in mechanisms. The methodology is evaluated in the context of a wireless handheld-device PL.

1. Introduction

Software product lines (PLs) are large-scale systems with high complexity, illustrating significant variability and a long life-span. These attributes, in combination with changes in the software domain and the underlying technology, impose a number of hard requirements on such systems. PLs must be changeable and evolvable and must support low effort, timely composition of flexible products.

Features play an important role in PLs. In this work features are "a logical unit of behavior that is specified by a set of functional and quality requirements and represent an aspect valuable to the customer" [4], [13]. Features are used for modeling the domain, managing variability, guiding future planning, as a communication basis between the system stakeholders or as a general guide for the system design.

Despite the crucial role of feature in PLs, the state of the art PL methods fail to provide a strong mapping between features and the software architecture. As a consequence we have the phenomenon of feature scattering and tangling as shown in Figure 1.

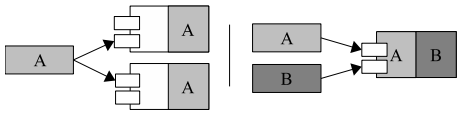

Figure 1. Feature A is scattered in two components (left). Features A and B are tangled in one component (right).

Changes in the PL occur mainly on the feature level. Changes in a PL come from the system stakeholders and the evolution of the underlying technology. Stakeholder changes are directly expressed in terms of the PL features. In a well designed feature model (FM), anticipated technology changes can be directly mapped to features. In any other case, changes may arbitrarily affect various parts of the system and respectively various PL features. From the above it becomes obvious that the effect of scattering and tangling of features has a large impact on PL maintainability.

Furthermore, a lot of the state of the art PL methods incorporate a manual approach to application engineering, thus requiring extra effort for product instantiation. Additionally, the implementation of variability in PL products, e.g. boot-time or run-time variability is mainly performed through the use of design patterns, introducing extra architectural entities. Because of feature scattering and tangling, these patterns are applied to numerous points in the system in order to allow for feature-level variability, which influ-

ences the system's architectural integrity [5]. The aforementioned issues deteriorate the time-to-market and cost-effectiveness of the PL.

Feature-Architecture Mapping (FArM) is a methodology developed to allow a stronger mapping between features and the architecture. FArM progressively transforms the initial PL FM to derive architectural components, which encapsulate the transformed features' business logic and whose communication reflects the feature interactions.

Section 2 illustrates the industrial case study used for the method's description and evaluation. Sections 3 to 4 introduce FArM and its processes. Section 5 presents the method's evaluation. Finally, sections 6 and 7 present related work, the conclusions and future work respectively.

2. MobilePL Product Line

The feasibility of FArM has been evaluated with a case study from the domain of IDEs [15]. For the purposes of further developing and evaluating FArM, a case study from the domain of distributed, embedded applications has been conducted [14]. Based on the experiences of this case study the method is currently being evaluated through the development of a PL from the domain of mobile phones. The PL is based upon the Symbian OS. The structure of the PL APIs is shown in Figure 2.

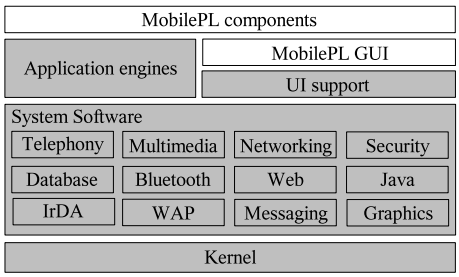

Figure 2. The MobilePL APIs (non-shaded).

In the MobilePL architecture, data are exchanged between the handheld and the MobilePL Enterprise Server. The latter is behind the corporate firewall and communicates with email and content servers as shown in Figure 3. A key feature of the MobilePL is the Push feature. This enables the automatic propagation of content (e.g. emails) when available, without need for an explicit request from the handheld user.

3. Overview of the Feature-Architecture Mapping (FArM) Method

FArM's goal is to provide a strong mapping between features in the FM and the PL architecture. In order to achieve

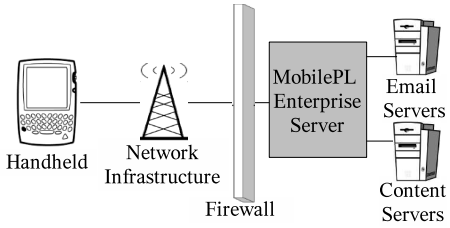

Figure 3. MobilePL system architecture.

this, FArM utilizes an initially constructed FM produced by a domain analysis method, e.g. FODA [8], to derive architectural components closely related to features. The initial FM is progressively transformed by two cooperating teams, namely, a feature analyst team and an architecture team. The feature analysts work on the FM-level, e.g. with feature specifications. The architecture team makes decisions for the component specification and the system architecture.

The FArM transformation flow is shown in Figure 4. The FArM transformations occur in many iterations. In each iteration the developers concentrate primarily in one transformation but previous transformations may also be revisited, in which case the following transformations must be performed again in the given order. Note also that architecture development takes place in each iteration. This iterative development model assures the consistence of the produced artefacts, i.e. FM and reference architecture and allows a balanced approach to design and implementation.

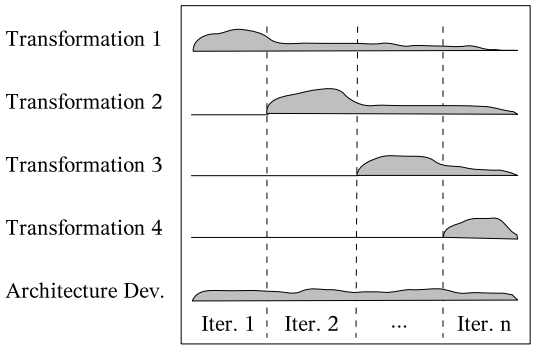

Figure 4. FArM transformation flow and iterations.

The FArM transformations handle:

Trans. 1. Non-Architecture-Related & Quality features

Trans. 2. Architecture Requirements

Trans. 3. Interacts Relations

Trans. 4. Hierarchy Relations

3.1. Initial Feature Model

The FM serving as input to FArM must satisfy the following elementary criteria:

- Features must be derived at least from a customer perspective. This criterion is a vital prerequisite for the development of maintainable systems. Nonetheless, features from other stakeholder perspectives may be included in the initial FM, e.g. from an architectural or managerial perspective.
- All features must be accompanied by a searchable specification. This can be in the form of text and/or models. This criterion enables the analysis of the features during the FArM transformations.
- The FM must possess basic structural properties, i.e. features must be structured in a hierarchical manner having feature-subfeature relations with the proper relation-cardinalities. Furthermore, features must be marked at least as being mandatory, optional or alternative. This criterion illustrates the basic relations between features and is also an indicator of a semantically correct initial FM.

The above listed criteria are satisfied by the majority of the existing domain analysis methods. As it can be seen, efforts have been made in FArM to allow for versatility in the application of domain analysis. Additional information included in the initial FM are properly handled throughout the FArM transformations.

3.2. Non-Architecture-Related & Quality Features

FArM strives to achieve a transformed FM where each feature can be implemented in an architectural component. Non-Architecture-Related (NAR) features have an insignificant impact on the software architecture and therefore cannot be mapped to an architectural component. Quality features influence the entire software system making it practically impossible for a direct implementation in an architectural component. In transformation 1, FArM handles NAR and quality features. The internals of these transformation are presented in sections 3.3 and 3.4

3.3. NAR Feature Resolution

NAR features in FArM belong to either of the following categories:

Physical Features representing physical attributes of the system. These features are resolved directly through hardware solutions

Business Features relating to pure business aspects of the system. These features are resolved through managerial solutions

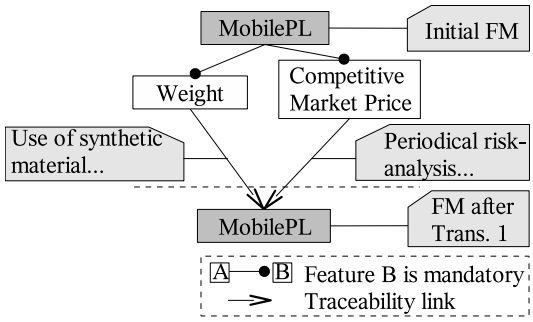

Figure 5. Resolution of NAR features.

In Figure 5 the **Weight** feature is a ***Physical*** feature and can be directly resolved through the use of synthetic material for the handheld's case and medium size. Physical NAR features should be carefully identified, since they may have a strong indirect influence of the software architecture. An example is **Battery**. At a first glance, this feature seems to be a physical feature. Nonetheless, battery duration imposes sophisticated power management algorithms having a non-trivial impact on the software architecture.

The **Competitive Market Price** feature of Figure 5 is a ***Business*** feature and can be resolved, e.g. through the managerial solutions of periodical risk-analysis or adding experts in the development team.

Note that the transformation decisions are captured in traceability links between the resolved features and the root feature (i.e. **MobilePL**) of the transformed FM. Traceability links are added throughout the FArM transformations to allow forward and backward traceability of the decisions made.

3.4. Quality Feature Resolution

Quality features are initially quantitatively defined through the creation of Profiles, as described in Bosch [4]. They are later resolved by resolving each part of their quantitative specification. For this purpose the elementary transformations of direct resolution, integration in existing functional features and addition of new functional features is used. More precisely, some parts of the specification can be directly resolved if they have no impact on the software architecture, like in the case of NAR features. Other parts are integrated into existing functional features by enhancing their specifications. Finally, new features can be added to provide functional solutions that fulfil the quality features' specifications.

Figure 6 shows the resolution of the **Security** quality feature of the MobilePL PL. One part of the quantitative specification of the **Security** feature is satisfied through the addition of the new functional feature **Firewall**. Other parts of the **Security** feature's specification are similarly resolved. For a more detailed discussion of this transformation see [16].

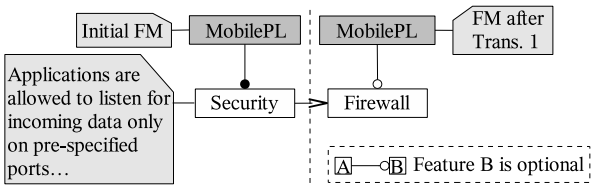

Figure 6. Resolution of the MobilePL Security feature.

3.5. Transformations based on Architectural Requirements

Commonly, domain analysis methods use domain experts or application documentation to analyze a domain. This leads to functional features expressing the customer's view of the PL. Nonetheless, there may exist hard architectural requirements that must be satisfied. Like in the case of quality features this is done through direct resolution, integration in existing functional features and addition of new functional features. An example is shown in Figure 7.

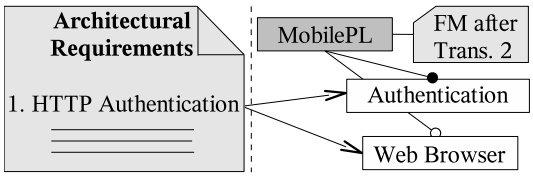

Figure 7. Resolution of the HTTP Authentication architectural requirement.

The HTTP Authentication architectural requirement refers to the authentication needed by various web sites. To reduce network traffic the architects add an **Authentication** feature to store the required usernames and passwords and automatically pass them to web sites needing authentication. If authentication fails, the challenge is sent back to the handheld and a prompt is shown in the web browser. Therefore, this part of the requirement specification is integrated into the pre-existing **Web Browser** feature.

3.6. Transformations based on Interacts Relations

After transformations 1 and 2, the FM contains exclusively functional features, both those from the customer point of view and those closer to the system architecture. FArM models the communication of features by introducing a new FM relation, namely, the interacts relation. Note that this term has already been used in related work [6] but it is explicitly extended in this work. A valid FArM interacts relation must belong at least to one of the following types:

Type 1. It connects two features where one feature uses the other feature's functionality

Type 2. It connects two features where the correct operation of one feature alters the behavior of the other feature

Type 3. It connects two features where the correct operation of one feature contradicts with the correct operation of the other

Figure 8 shows examples for each type of interacts relation. **Email** uses the functionality of **Push** to receive automatically propagated emails. The presence of the **Profiles** feature causes the **Sound** feature to reduce the ringing tone when the "silent" profile is active. The presence of the **Firewall** feature prevents the **Web Browser** from downloading non-authorized executable files.

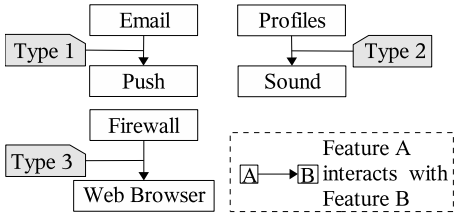

Figure 8. FArM interacts relations.

Initially, all interacts relations between features are identified. Features are then transformed based on FArM transformation criteria and finally, interfaces are assigned to the respective architectural components. The FArM transformation criteria are:

Criterion 1. The type of interacts relations

Criterion 2. The number of interacts relations

Based on the above criteria a feature can be integrated into another feature or a new feature can be added. An example for a transformation that may occur based on the type of interacts relations (criterion 1) can be seen in the **Firewall-Web Browser** relation. This relation belongs to the

second type of interacts relations, where the correct operation of the **Firewall** feature contradicts with the correct operation of the **Web Browser** feature when the latter blocks the download of executable files. In this case the developers may consider either integrating the **Firewall** feature in the **Web Browser** feature, which would resolve the contradiction or enhance the **Web Browser** feature by providing an open interface for control over the download process. Since the **Firewall** feature presents an important feature in the system and it also interacts with other features, the developers select the enhancement of the **Web Browser** feature.

An example of a transformation based on the number of interacts relations (criterion 2) is shown in Figure 9. In this example the **UI Support** feature has a large number of interacts relations, namely, with every feature requiring user interaction. For instance, the **Email** feature must present a GUI to the user to allow the reading of emails. Additionally, the **UI Support** feature has no extra logic in itself, i.e. it merely provides references to the UI-classes. For this reason the **UI Support** feature is integrated into every feature needing access to user interface functionality.

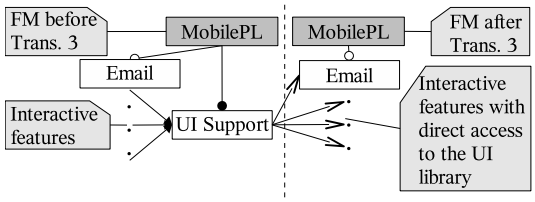

Figure 9. FM transformation based on the number of interacts relations.

3.7. Transformations based on Hierarchy Relations

Features within a hierarchy relation in the FM illustrate a strong logical relation. If features in a FM hierarchy can be mapped to components, one could use the hierarchy relations for the specification of the component interfaces. Components implementing a super-feature may serve as a central access point for functionality from and to components implementing the sub-features. For example, components beyond the hierarchy relation could access functionality provided by the sub-feature-components centrally, through the super-feature-component or the sub-feature-components could access the functionality of other features through the super-feature-component. This allows higher encapsulation and lower coupling between the various components of the PL. Additionally, super-feature-components could also be used as a switch mechanism between sub-feature-components, thus inherently supporting variability, e.g. the strategy design pattern can be implemented within the super- and sub-feature-components, without need for introduction of extra architectural entities.

Based on all the above, a valid FArM hierarchy relation must belong at least to one of the following types:

Type 1. The sub-feature specializes the super-feature

Type 2. The sub-feature is a part of the super-feature

Type 3. The sub-features present alternatives to their super-feature

Examples of valid FArM hierarchy relations are shown in Figure 10. The **Messages** - **Email**, **SMS** hierarchy relation belongs to both types 2 and 3 of the FArM hierarchy relations, since **Email** and **SMS** are both parts of **Messages** and present message alternatives. This hierarchy relation results to the addition of interfaces shown in the lower part of Figure 10. The *Messages* component provides a `send()` method for sending emails, which is used by the *Email* component. The **Encryption** - **Triple DES** hierarchy relation belongs to type 1 of the FArM hierarchy relations. As shown in the collaboration diagram of Figure 10, the *Triple DES* component specializes the *Encryption* component by providing a stronger encryption algorithm.

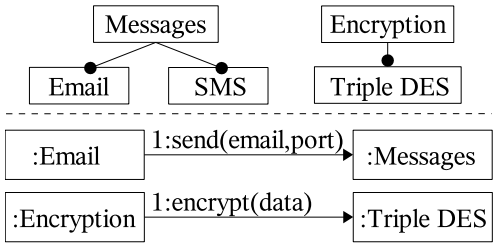

Figure 10. FM (upper part) and architectural implementation (lower part) of the FArM features.

This transformation starts by examining all features if they can initiate new sub-features. Then, all hierarchy relations are checked against the FArM hierarchy types for validity. Invalid hierarchy relations are removed and their features remain single, i.e. without any sub-features. Finally, new hierarchy relations are created according to the above types. In the last step, interfaces are added to the respective architectural components.

A representative example for this transformation is shown in Figure 11. The single features **Email** and **SMS** are identified as alternatives. The new **Messages** feature is added as their super-feature, thus complying with type 3 of the FArM hierarchy relations.

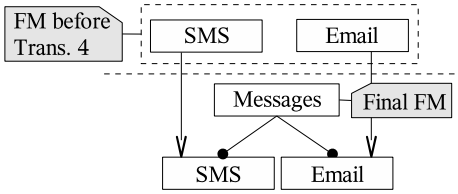

Figure 11. Addition of the Messages super-feature after the hierarchy transformation.

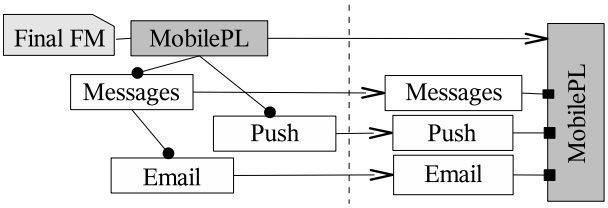

Figure 12. The final FM and plug-in architecture.

4. Architecture Development

In the last iterations of the hierarchy transformation the final transformed FM is developed. The FM contains only functional features. Each feature has a concrete specification regarding its responsibilities. The communication between features has been modelled through interacts and hierarchy relations. For each of the transformed features a respective architectural component has been derived encapsulating the business logic of the feature and having an interface that reflects the feature communication.

As shown in Figure 4, architecture development takes place in each FArM iteration. Throughout the transformations, architecture development occurs mainly in the form of component specification. In the last FArM iterations the system architects commit to a certain architecture. That is, they adopt a specific architectural style and the developed components are adapted to that style, e.g. the decision to use a layered architecture would lead to ordering of each of the components in a specific layer.

During component adaptation, FArM takes into consideration two important points of PLs, namely, product composition and flexibility. These issues are addressed through the explicit integration of plug-in mechanisms in the developed components. For example, the use of XML technology to impose a plug-in protocol, the introduction of a class-loader mechanism or the use of a dynamic architectural style, e.g. the Kernel architectural style.

A partial view of the MobilePL architecture developed during this case study is shown in Figure 12. The final transformed FM contains features mapped to an architectural component. The **MobilePL** root feature implements the plug-in platform where each component is plugged.

The class diagram of Figure 13 shows the facade classes of the respective components in Figure 12. The methods `getFeature()`, `exit()` and `load()` are part of the plug-in API. The methods `listen()`, `recv()` and `newThread()` have been added during the transformations as a result of component communication.

Figure 14 shows the collaboration diagram for the reception of a pushed email. The *Push* component receives a request for listening to a certain port for incoming data. The

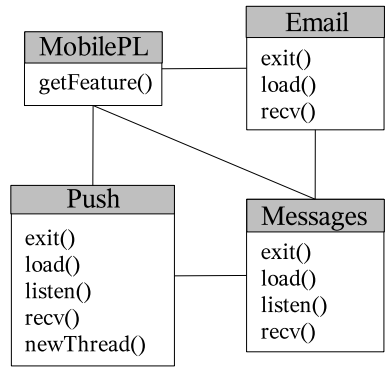

Figure 13. Class diagram of the facade classes of FArM components.

request has been sent originally from the *Email* component through the *Messages* component implementing the super-feature of the **Email** feature. The *Push* component starts a new listening thread and returns the thread id. Upon the arrival of data on the specified port, *Push* sends the data to the *Messages* component, which in turn propagates the data to the *Email* component.

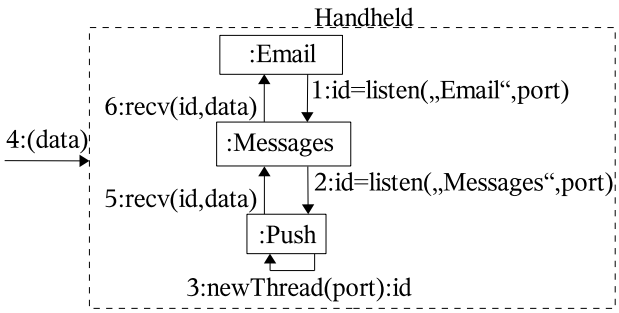

Figure 14. Collaboration diagram for the reception of a pushed email.

5. FArM Evaluation

This section provides an evaluation of FArM regarding the maintainability and flexibility of the produced PL. It also gives references to the current tool support for the method.

5.1. Maintainability

As mentioned in section 1, PL changes are mainly driven from new market needs and technology enhancements both occurring on the feature level. The following sections will discuss the addition and changing of a feature.

5.2. Addition of a feature

FArM supports the addition of a feature by carrying out the aforementioned transformations for the new feature. The latter is exposed to the elementary transformations, being directly resolved, integrated into existing features or implemented into new functional features. Its communication-needs to other PL features are modelled and the possible newly identified components are derived. Finally, the plug-in nature of the FArM architecture allows the easy integration of new components through a plug-in mechanism.

An example is the addition of the optional **MMS** feature. The feature is first added to the initial FM as a single feature. The feature remains intact through transformation 1 since it is neither a quality nor a NAR feature. In transformation 2 the architects identify the need to extend the communication protocol between the server and the handheld to handle multimedia content. This architectural requirement is integrated into the pre-existing **Messages** feature. In transformation 3 the need for sending messages causes the addition of an interacts relation between the **MMS** and the **Messages** features. In the last transformation the interacts relation is transformed into a hierarchy relation between **Messages** and **MMS**. Finally, the respective architectural component is adapted to the plug-in protocol of the MobilePL architecture through the addition of interfaces like `load()` and `exit()`, for loading the feature during system booting and freeing of resources during system shutdown.

5.3. Changing of a feature

The following change scenario is considered: "*Simultaneous data reception of many push-enabled applications causes performance deterioration on the handheld device. A prioritized policy for pausing and resuming pushed data shall be imposed...*". In the FArM architecture (Figure 14) only the *Push* component needs to be changed to assign priorities to support the pausing and resuming of the listening threads. The *Push* component can inform the server for pausing and resuming data transmission respectively. As shown from this scenario the encapsulation of the feature business logic in one architectural component has localized the effect of the change, thus increasing system maintainability. Even in the case where a large number of features needs to be changed, the FArM interacts relations can be used to estimate and guide the development efforts.

Figure 15 shows the Blackberry implementation of the **Push** feature. Blackberry [10] is a commercial PL of the RIM company. The logic of the Blackberry **Push** feature is scattered to all push-enabled components of the system. Each push-enabled component has to create its own thread and listen to a specified port for incoming data. The realization of the previous change scenario for the Blackberry **Push** feature would cause the changing of each push-enabled component, plus the addition of a priority manager to synchronize all components.

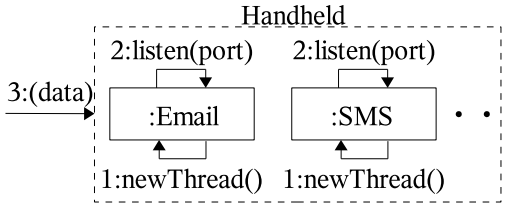

Figure 15. Blackberry implementation of the Push feature.

5.4. Flexibility

With the term flexibility, this work refers to the effort needed for product instantiation and the efficiency with which feature-level variability is added to the PL products.

Many PL methods necessitate extra effort for the instantiation of PL products. Product instantiation takes place, e.g. through extension of the developed components during PL engineering. Other methods require the writing of "glue-code" for the connection of the developed components. Due to a stronger mapping between features and the architecture, the effort for the composition of PL products can be reduced. Throughout the FArM transformations traceability links are added between the transformed FMs and the final FM and PL architecture. Because of this fact, the selection of a feature in the initial FM by a customer can be traced down to the transformed FM and the respective architectural component(s). Additionally, the explicit integration of plug-in mechanisms in the PL architecture reduces the effort for the composition of the final product.

The other issue discussed under the term flexibility is that of feature-level variability. As mentioned in section 1, the implementation of feature-level variability in PL prod-

ucts, has a negative impact on the system's conceptual integrity [5], because of the introduction of extra architectural entities, baring little or no relation to neighbor system components.

FArM addresses this issue with a number of rules applied throughout the FM transformations. Namely, features with boot-time or run-time variability should not be integrated into other features. This assures that the business logic of these features is implemented in one or more architectural components, which can later be used for the implementation of a variability mechanism. During transformation 3, the number of interacts relations of variable features must be minimized, e.g. through the addition of new features that are assigned some of the variable feature's interacts relations. This encapsulates the variable feature and allows for an easier implementation of the variability mechanism. In transformation 4, the super-features of variable features must be extended to receive the needed variability mechanism. For example, the *Messages* component should be extended with the architectural entities of the *strategy* design pattern, in order to properly propagate received messages to one of the *Email*, *SMS* or *MMS* components.

5.5 Tool Support

A prototype tool for research on the efficient application of the FArM method has been developed with the *DOME* (DOmain Modeling Environment) tool. DOME is a metacase system suitable for building object-oriented software models. FArM specifics, such as traceability links, interacts relations and constraints have been implemented as a DOME tool-specification. This is a graphical, high-level specification of node and connector types, connection constraints, and additional syntax and semantics.

For the industrial application of FArM though, a collection of professional tools can be used. For the domain analysis and transformations of FArM the developers may use a common industrial documentation managing tool like IBM's *Rational Requisite Pro*. This will allow the capturing and tracing of requirements, feature specifications, as well as the various transformation decisions. The design and implementation of the PL architecture can be done with conventional case tools, e.g. Borldand's *Together* tool-suite. Finally, the generation of end-products can be implemented with the use of a software dependencies and packaging tool, e.g. the *RPM* Package Manager. More precisely, each component implementing a feature can be placed in a separate package and the FM requires and excludes relations can be applied in the form of package dependencies.

Currently, work is done on the unified application of the FArM processes within the *Eclipse* environment, through the use of the provided plug-in interfaces.

6. Related Work

This section illustrates the main state of the art PL methods from the perspective of feature-architecture mapping. It also presents related work to the internals of the FArM processes.

The main state of the art PL methods are FeatuRSEB [7], KobrA [1] and Functionality-based Architectural Design (FAD) [4]. FeatuRSEB and KobrA are based on use-cases to derive architectural components. In order to provide a mapping between features and architecture, FeatuRSEB makes use of *traces*, while KobrA introduces a similar construct called *decision model*. The derivation of PL components through use-cases, promotes the phenomenon of feature scattering and feature tangling as described in section 1. A feature is described in parts of various use-cases. Consequently, deriving components from use-cases eventually leads to the implementation of features within numerous architectural elements. Nevertheless, use-case derived architectural components illustrate high encapsulation and low coupling on the use-case level, allowing for sufficient maintainability in various domains. The use of traces and decision models for the mapping between features and the architecture is indeed a sufficient solution for medium sized PLs. Unfortunately, a large number of features and classes in a system lead to an explosion of the number of traces and the number of decisions for the decision model. This makes the creation and maintenance of these constructs a challenging task.

FAD is based on the concept of archetypes to derive architectural components. Archetypes represent the core abstractions of a system. During FAD the identified archetypes are instantiated through the use of architectural styles and design patterns. FAD provides no mapping mechanism between features and the architecture. The main issues with FAD are the identification of archetypes, their instantiation and of course, the lack of a mapping mechanism. FAD does provide a number of hints to support archetype identification, although it denotes that it is mainly a creative, intuitive process. The instantiation of archetypes through architectural styles and design patterns also leads to the introduction of potentially "artificial" entities, offending the principle of the system's conceptual integrity [5]. Nonetheless, FAD archetypes present re-occurring architectural patterns, contributing to overall conceptual integrity.

An attempt to map the FODA domain model to a generic architecture has been made in [12]. This method utilizes FODA's information model to derive objects that map to the constructs of the Object Connection Architecture (OCA) [9]. The latter is an architectural model focusing on the separation of control and data flow of large software systems. This approach increases the system's flexibility and maintainability for numerous scenarios. Nonetheless, objects in

this method are derived from FODA's information model, rather than the FM. The information model captures and defines the domain knowledge and data requirements that are essential in implementing applications in the domain. This fact results to architectural components partially implementing various customer-related features, which in turn leads to a poor feature-architecture mapping with a maintainability penalty.

On their attempt to compensate for limited PL flexibility a number of generative programming techniques have been integrated into PL methods. Representative examples are the HyperFeatuRSEB method [3] integrating the Hyperspace approach [11] with FeatuRSEB and the GenVoca method [2]. These methods are based on programming techniques supported by UML extensions to logically map code to features. This approach does enhance the PL flexibility, but on the expense of PL maintainability. More precisely, the extra models and code introduced increase the effort needed to accommodate changes and support PL evolution.

Related work to the inner processes of FArM can be found in FAD [4] regarding quality feature resolution. In FAD quality features are resolved through architectural styles and design patterns. Although this approach achieves quality feature resolution, it also introduces architectural entities potentially breaking the system's conceptual integrity. FArM makes use of design patterns only for the internal parts of the derived components, where size and complexity remain manageable. Work has been done for FArM's type 3 of interacts relations [6]. This work assumes that components operate with no knowledge of each other. FArM explicitly models feature interactions and extends the concept of interacts relations. Nonetheless, the techniques of the related work can be used in FArM, as long as the principle of conceptual integrity is taken into consideration.

7. Conclusions and Future Work

This paper introduced the Feature-Architecture Mapping (FArM) method for feature-oriented development of software PLs. FArM utilizes a number of mature software practices, enriched with new elements to achieve a stronger mapping between features and the architecture.

By iteratively refining the initial FM, a FM is constructed, containing exclusively functional features, whose business logic can be implemented into architectural components. Furthermore, the FM hierarchy and the FArM interacts relations model the communication between the PL features. This leads to a PL architecture where each architectural component encapsulates the business logic of a feature and the component communication reflects the feature interactions. The evaluation of the method has shown its potential to enhance PL maintainability and flexibility.

Further work in FArM includes the concrete definition of its applicability in other application domains and the authoring of a detailed report for its implementation.

References

[1] C. Atkinson et al. *Component-based Product Line Engineering with UML*. Addison-Wesley, 2002.

[2] D. Batory and J. Geraci. Composition Validation and Subjectivity in GenVoca Generators. *IEEE Transactions on Software Engineering*, 23(2):67–82, 1997.

[3] K. Boellert. *Object-Oriented Development of Software Product Lines for the Serial Production of Software Systems*. PhD thesis, TU-Ilmenau, Ilmenau Germany, 2002. german.

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

[5] F. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Longman, 1995.

[6] M. Calder, M. Kolberg, M. Magill, and S. Reiff-Marganiec. Feature Interaction A Critical Review and Considered Forecast. *Elsevier: Computer Networks*, 41(1):115–141, 2003.

[7] D. Griss, R. Allen, and M. d'Allesandro. Integrating Feature Modelling with the RSEB. In *5th International Conference of Software Reuse*, 1998.

[8] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Software Engineering Institute, Carnegie Mellon University, 1990.

[9] K. J. Lee, M. J. Rissman, R. DIppolito, C. Plinta, and R. V. Scoy. An OOD Paradigm for Flight Simulators, 2nd ed. Technical report, Software Engineering Institute, Carnegie Mellon University, Sept. 1988.

[10] R. I. M. Limited. *BlackBerry Java Development Environment Version 3.6 Developer Guide Volume 1, Fundamentals*. RIM, 2003.

[11] H. Ossher and P. Tarr. *Software Architectures and Component Technology*, chapter Multi-Dimensional Separation of Concerns and the Hyperspace Approach. Kluwer Academic Publishers, 2001.

[12] A. S. Peterson and J. Jay L. Stanley. Mapping a Domain Model and Architecture to a Generic Design. Technical report, Carnegie Mellon University, Software Engineering Institute, 1994.

[13] M. Riebisch. Towards a More Precise Definition of Feature Models. *Workshop at ECOOP*, pages 64–76, 2003.

[14] P. Sochos. Mapping feature models to the architecture. *First International SPLYR Workshop*, pages 51–60, 2004.

[15] P. Sochos, M. Riebisch, and I. Philippow. Feature-oriented development of software product lines: Mapping feature models to the architecture. *Net.ObjectDays*, pages 138–152, 2004.

[16] P. Sochos, M. Riebisch, and I. Philippow. Feature-oriented architecture design for maintainability and evolution of product lines. *Software Engineering*, March 2005. german.