THE FEATURE-ARCHITECTURE MAPPING METHOD FOR FEATURE-ORIENTED DEVELOPMENT OF SOFTWARE PRODUCT LINES

Dissertationsschrift

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

VORGELEGT DER FAKULTÄT FÜR INFORMATIK UND AUTOMATISIERUNG
DER TECHNISCHEN UNIVERSITÄT ILMENAU

von **MSc Periklis Sochos**
geboren am 17. Dezember 1978 in Athen, Griechenland

vorgelegt am 20.9.2006

wissenschaftliche Aussprache am 25.04.2007

Gutachter:

1. PD Dr.-Ing. habil. Matthias Riebisch
2. Univ.-Prof. Dr.-Ing. habil. Wolfgang Fengler
3. Prof. Dr. rer. nat. Ralf H. Reussner

urn:nbn:de:gbv:ilm1-2007000087

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

Dissertation

for the attainment of the academic degree of
Doktoringenieur (Dr.-Ing.)

SUBMITTED AT THE FACULTY OF INFORMATICS AND AUTOMATION
OF THE TECHNICAL UNIVERSITY OF ILMENAU

by **MSc Periklis Sochos**
born on the 17th of December 1978 in Athens, Greece

submitted on the 20.9.2006

convocation on the 25.04.2007

Examiners:

1. PD Dr.-Ing. habil. Matthias Riebisch
2. Univ.-Prof. Dr.-Ing. habil. Wolfgang Fengler
3. Prof. Dr. rer. nat. Ralf H. Reussner

urn:nbn:de:gbv:ilm1-2007000087

Abstract

Software product lines are the answer of software engineering to the increasing complexity and shorter time-to-market of contemporary software systems. Nonetheless, software product lines demand for advanced maintainability and high flexibility. The latter can be achieved through the proper separation of concerns. Features pose the main concerns in the context of software product lines. Consequently, one feature should ideally be implemented into exactly one architectural component. In practice, this is not always feasible. Therefore, at least a strong mapping between features and the architecture must exist. The state of the art product line development methodologies introduce significant scattering and tangling of features. In this work, the Feature-Architecture Mapping (FArM) method is developed, to provide a stronger mapping between features and the product line architecture. FArM receives as input an initial feature model created by a domain analysis method. The initial feature model undergoes a series of transformations. The transformations strive to achieve a balance between the customer and architectural perspectives. Feature interaction is explicitly optimized during the feature model transformations. For each feature of the transformed feature model, one architectural component is derived. The architectural components implement the application logic of the respective features. The component communication reflects the feature interaction. This approach, compared to the state of the art product line methodologies, allows a stronger feature-architecture mapping and for higher variability on the feature level. These attributes provide higher maintainability and an improved generative approach to product instantiation, which in turn enhances product line flexibility. FArM has been evaluated through its application in a number of domains, e.g in the mobile phone domain and the Integrated Development Environment (IDE) domain. This work will present FArM on the basis of a case study in the domain of artificial Neural Networks.

Kurzfassung

Software Produktlinien sind die Antwort von Software Engineering auf die zunehmende Komplexität und kürzeren Produkteinführungszeiten von heutigen Softwaresystemen. Nichtsdestotrotz erfordern Software Produktlinien eine fortgeschrittene Wartbarkeit und hohe Flexibilität. Das kann durch die angemessene Trennung der Belange erreicht werden. Merkmale stellen die Hauptbelange im Kontext von Software Produktlinien dar. Demzufolge sollte ein Merkmal idealerweise in genau einer Architekturkomponente implementiert werden. In der Praxis ist das jedoch nicht immer machbar. Deshalb sollte zumindest ein starkes Mapping zwischen Merkmalen und der Architektur bestehen. Die Methoden zur Entwicklung von Software Produktlinien, die dem Stand der Technik entsprechen, führen zu bedeutender Verstreutheit und Vermischung von Merkmalen. In dieser Arbeit wird die Feature-Architecture Mapping (FArM) Methode entwickelt, um ein stärkeres Mapping zwischen Merkmalen und der Produktlinien-Architektur zu erzielen. Der Input für FArM besteht in einem initialen Merkmalmodell, das anhand einer Methode zur Domänenanalyse erstellt wurde. Dieses initiale Merkmalmodell wird einer Serie von Transformationen unterzogen. Die Transformationen streben danach, ein Gleichgewicht zwischen der Sichtweise von Kunden und Softwarearchitekten einzustellen. Die Merkmalinteraktionen werden während der Transformationen ausdrücklich optimiert. Von jedem Merkmal des transformierten Merkmalmodells wird eine Architekturkomponente abgeleitet. Die Architekturkomponenten implementieren die Applikationslogik der entsprechenden Merkmale. Die Kommunikation zwischen den Komponenten spiegelt die Interaktion zwischen den Merkmalen wider. Dieser Ansatz führt im Vergleich zu den Produktlinien-Entwicklungsmethoden des Stands der Technik zu einem stärkeren Mapping zwischen Merkmalen und der Architektur und zu einer höheren Variabilität auf Merkmalebene. Diese Eigenschaften haben eine bessere Wartbarkeit und eine vereinfachte generative Produktinstanzierung zur Folge, was wiederum die Flexibilität der Produktlinien steigert. FArM wurde durch ihre Anwendung in einigen Domänen evaluiert, z.B. in den Domänen von Mobiltelefonen und Integrierten Entwicklungsumgebungen (IDEs). Diese Arbeit wird FArM anhand einer Fallstudie in der Domäne von Künstlichen Neuronalen Netzwerken präsentieren.

Acknowledgements

I would like to thank my supervisors Prof. Dr.-Ing. habil. Ilka Philippow and Priv.-Doz. Dr.-Ing. habil. Matthias Riebisch for their guidance throughout this work, as well as the federal state of Thüringen for the financial support through the granting of the Landesgraduierten scholarship. A big thank goes also to my professor and supervisor during my studies in Greece Prof. Dr.-Ing. Basilios Spiropoulos.

Last but not least, I want to thank my family for their loving support that made this work possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The ever growing demand for innovative products and the hard competition in the industrial arena constantly increase the size and complexity of software, requiring shorter times-to-market, which respectively raise the demand for higher flexibility. This trend is evident in almost every domain in today's software industry, e.g. in the telecommunications, automotive or logistics domain. For instance, mobile phone vendors must support many different protocol standards, a wide variety of functional features and capabilities, different user interface designs and many platforms and environments. Additionally, they must evolve their product palette in very short periods of time to provide online gaming or photography capabilities, push-enabled applications and UMTS technology.

Standard software development methodologies are not adequate for the construction of such software systems. On one hand, the size and complexity of the software is accompanied by significant variability that must be implemented, while on the other hand, there exists a lot of commonality that must be exploited. Furthermore, the one-product-at-a-time development model of standard methods is not compatible with the constantly decreasing time-to-market and the need for flexible, customizable products.

A solution to these issues is the prefabrication of software building blocks. More precisely, the common functionality of a domain is prefabricated in a core-set of software building blocks and the variable functionality in another set. The development of products takes place through the combination of building blocks. This is the so called software product line (PL) approach, where the building blocks are software components and the developed software platform is the PL architecture.

Nonetheless, adopting a PL approach alone does not completely solve the aforementioned issues. Two crucial preconditions must be satisfied. The PL components must be relatively stable with respect to changes and the product instantiation process must take place efficiently.

The stability of the PL components plays a main role in the maintainability and evolvability of the software system. Ideally, changes in one part of the software should not propagate, rather, they must have a local impact. Stability can be achieved if the PL architecture adheres to the principles of separation of concerns [Par72], high cohesion and low coupling. In the context of PLs, features pose the main concerns (sect. 2.1). Therefore, a higher stability in a PL can be achieved, if the PL architecture adheres to the principles of high cohesion and low coupling on the feature level.

The other precondition that must be satisfied is the efficient instantiation of PL products. This has a direct influence on the time-to-market. In the context of PLs, products are defined in terms of features. Therefore, the PL architecture must enable the composition of products based on features with minimal effort. In other words, the PL architecture must allow the efficient application of variability mechanisms on the feature level.

Ideally, in order to achieve high cohesion, low coupling and variability, all on the feature level, one PL feature should be implemented in exactly one architectural component. This way, the stakeholder concerns would be perfectly encapsulated into self-contained constructs, which could be arbitrarily combined for the instantiation of a PL product.

Unfortunately, this is not feasible with today's technology (chapter 2). Instead, a stronger mapping between features and the architecture is needed. An architectural component should at least encapsulate the application logic of one feature and provide an interface for the needed feature interaction. This way, changes will either remain local within one architectural component or at most propagate to components implementing interacting features. This would lead to higher system stability. Additionally, variability mechanisms can be directly applied to components that implement the PL features, thus allowing for efficient variability on the feature level. PL product instantiation can then be reduced to the inclusion or exclusion of components. These two factors would lead to higher flexibility and shorter time-to-market.

Goal of this work is to develop a methodology for the enhancement of the mapping between PL features and the PL architecture. The methodology will provide an iterative approach for the derivation of a software architecture based on customer-specific features. An initial feature model developed during the domain analysis will serve as input to the method and will be iteratively transformed. Throughout these transformations, existing features are enhanced or merged with each other and new features are added, so as to achieve a balance between the customer and architectural perspectives. The final transformed feature model will hold only functional features, who's application logic can be directly implemented in exactly one architectural component. Furthermore, the feature interactions will also be reflected on the component interfaces and their communication. Traceability links will be utilized to connect the initial features to their transformed descendants, thus allowing for a stronger mapping between customer features and the PL architecture, which respectively leads to higher flexibility and shorter time-to-market.

Structure of the Work

Chapter 2 examines the state of art methods from the perspective of feature-architecture mapping, feature-level variability and PL product instantiation. It also lays out a concrete plan for the enhancement of feature-architecture mapping. Other works related to or used by the proposed solution are presented at the end of chapter 2. Chapter 3 gives insight to the case study used for the description and evaluation of the methodology developed in this work. Chapter 4 delves into the details of the proposed methodology. Finally, chapters 5 and 6 respectively validate the achievement of a stronger feature-architecture mapping and conclude the work.

Chapter 2

State of the Art

Chapter 1 denoted the importance of a strong feature-architecture mapping. The latter results in higher stability, more efficient feature-level variability and product instantiation. This chapter will examine the state of the art approaches to the aforementioned issues. In order to provide a solid basis for further discussion in the context of PLs, an introduction to the main PL concepts is given. Following this, the most representative PL methods will be selected and examined from the perspective of feature-architecture mapping, feature-level variability and product instantiation. Other related technologies that contribute to these issues are also examined. The identified problems of the state of the art approaches will then serve as an input to the conception of a plan for their resolution. Works used in the proposed solution are discussed at the end of this chapter.

2.1 Software Product Lines

The concept of PLs has emerged during a long process towards large-scale reuse of software. The vision of large-scale reuse has its roots in other engineering domains, e.g. architecture of buildings or the automobile industry. In these domains, products are composed from a well predefined set of components in a clearly prescribed way, thus leading to large-scale reuse. This allowed the partial, if not complete standardization of the production process and thus to significant cost and quality benefits.

Software development was initially performed on a one-product-at-a-time basis. A customer was able to either build an individual product or buy a standardized product, whereby both alternatives bared their own risks. On the one hand, projects for the development of an individual product carry a high risk of failure, extreme costs and poor quality, while on the other hand, standardized products may only partially cover the customer requirements, but provide high quality. From the reuse point of view, individual software provided primarily low-level reuse, e.g. code reuse or reuse of fine-grained library functions. In the majority of the projects for individual products, a constant "reinvention of the wheel" was required. Standardized software per definition achieves large-scale reuse, i.e. the product is sold as is with a few customization possibilities.

A big step to large-scale reuse was seen in the turn towards a domain-based approach to software engineering. The concept of PLs was born. A definition is given in [CN01]:

"*A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.*"

PLs provide large-scale reuse, since they are build from a common set of core assets. The core assets are developed for optimal reuse, based on the commonalities and variabilities of the domain. A PL product has to either manually be constructed from the set of core assets or it can be generated from the assets. In both cases, the time-to-market of the product is relatively short. PLs require a high initial investment, which increases the development costs. Nonetheless, the investment can be compensated through the instantiation of a small number of PL products. From the above it becomes clear that PLs provide, along with large-scale reuse, a flexible, cost-effective solution, compared to traditional development approaches.

The success of PLs is evident in today's software industry. Representative examples can be found in the Software Engineering Institute [SEI05] Hall of Fame. Nominated PLs are the Diesel engine PL, by Cummins Inc., which implements the software to micro-control ignition in order to produce an optimum mix of power, economy and emissions. Furthermore, the Bold Stroke avionics software family provides a wide range of artifacts required to create Operational Flight Programs for a variety of Boeing military fighters. Finally, another nominated PL is the Nokia Mobile Phones PL.

Development Process

The development process of PLs is shown in fig. 2.1. PL development is divided into domain engineering and application engineering. The iterative nature of the PL development processes is indicated through the arrowed circles. Domain engineering is responsible for the creation of the core assets. Application engineering is responsible for the development of the PL products from the core assets.

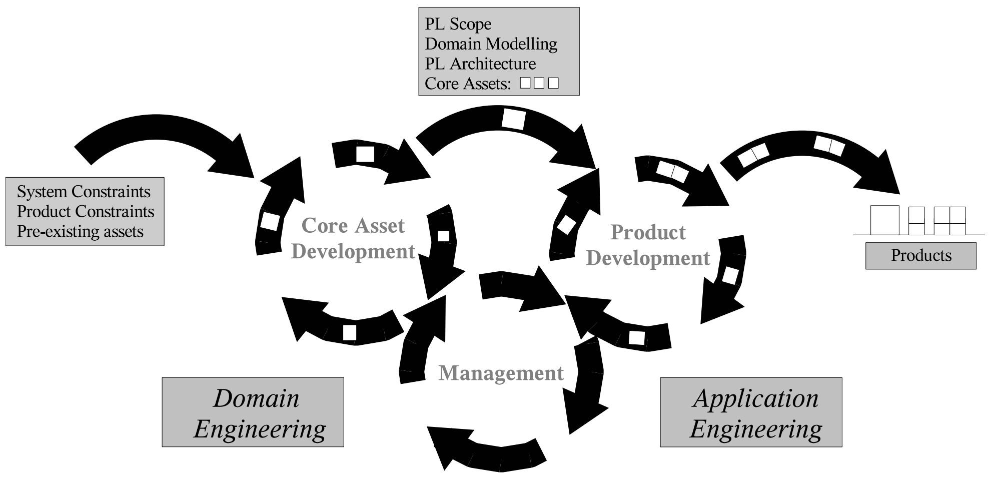


Figure 2.1: A common development process of a PL

PL engineering receives as input the system and product constraints, as well as pre-existing assets, e.g. legacy code. During domain engineering, scoping is performed for the definition of the PL limits, i.e. what should be implemented by the PL and what is not covered. Then follows the domain modeling. In this stage, the commonalities and variabilities of the domain are captured and documented. Afterwards begins the development of the common PL architecture. This includes the development of components, sub-systems or reusable packages.

In application engineering, the PL engineering artifacts are reused for the development of applications. This process covers customer requirement analysis and comparison to the PL capabilities. If the customer requirements are not covered by the PL, an instantiation of the core PL architecture is specialized and, depending on the importance of these new requirements for the future versions of the PL, the core PL architecture can also be extended.

Role of Features

A very important concept in the development of PLs is that of *features*. Features play a central role in the state of the art PL methods. This section briefly presents and motivates the use of features. There exist a number of definitions for features. This work adopts a combination of two representative definitions from [Bos00] and [Rie03].

"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".

Features are most commonly used for domain modeling. The modeling of a domain requires concise terms for the expression of complex domain entities. On the one hand, feature names consist of just a few keywords allowing a compact means of expression. On the other hand, feature specifications may contain a complete description of the underlying concepts. This can be achieved in a variety of ways, e.g. through free-text, use-case models, etc.

Additionally, the variability of a domain can be captured through the use of features. During domain analysis, the common features of the domain products are identified. Features belonging to individual products present the variability points of the domain. Features may also model the binding-time of variability. That is the point of time in which a feature is integrated into the product. This can range from compile-time, load-time to run-time variability.

Because of their property to model domain entities and their variability, features can also be used for the instantiation of PL products. A customer may select a number of PL features to be included in his/her personal product. For this purpose, features also model the various constraints of product composition. In the simplest case, a feature may be optional or mandatory and it may require or exclude other features.

During domain modeling, the above information is captured in the so called *feature model* (FM). The latter can have the form of a list of features, of a feature graph or a combination of both. There exist numerous notations for FMs that include a variety of information.

An example of a basic FM is shown fig. 2.2. The **MobilePL** feature is the root feature of the FM representing the PL. **Messages** and **Network** are two mandatory features of the PL. This means that they must be included in every PL product. These features have the optional features **MMS**, **WAP** and **HTTP** as sub-features respectively. The **MMS** feature has a requires relation to **WAP**. This indicates the fact that a customer selecting the **MMS** feature must also select the **WAP** feature. In this implementation of the PL, **MMS** also excludes the **HTTP** feature. This indicates that a customer selecting the **MMS** feature is not allowed to select the **HTTP** feature.

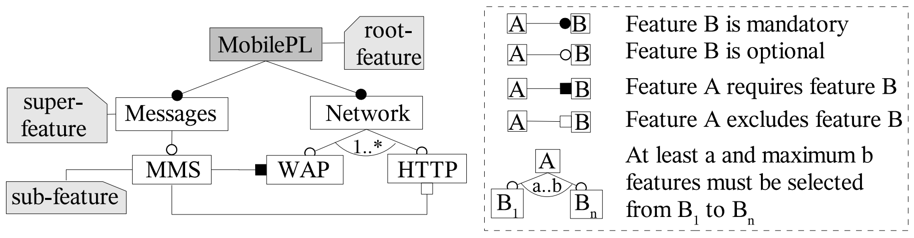

MobilePL
root-feature
super-feature
Messages
Network
1..*
MMS
WAP
HTTP
sub-feature
Feature B is mandatory
Feature B is optional
Feature A requires feature B
Feature A excludes feature B
At least a and maximum b features must be selected from B1 to Bn
a..b

Figure 2.2: A basic FM of a mobile phone PL

Features can also be effectively used to support the communication between the system stakeholders. This is very important for a harmonious cooperation and for the effectiveness of the works processes. Features provide a communication basis between the various stakeholders of a system, from PL customers, to marketing personnel and designers, down to the system programmers. This is due to the intuitive nature and compact description of features, accompanied with specific information for each party. Based on features, one can perform future planning. That is, the marketing department of a PL can identify trends in a market segment based on customer wishes. The latter can be easily expressed in terms of features. This information can then flow into the FM and create a base on which the PL developers can plan the future versions of the PL.

Occasionally, features also serve as a guide for the system design. Since features pose the main market drivers for the PL, designing the PL based on features can lead to benefits in the system's maintainability and deployment. Additionally, a number of useful design entities can be inspired from domain concepts.

2.2 Product Lines Methods

The PL methods examined in this work are selected based on a variety of criteria. Namely, their chronological order, their acceptance in the software industry and the volume of available documentation. Fig. 2.3 is originally composed from [Boe02] and [Str03] and has been updated to show the historical evolution of PL methods. The most recent PL methods are shown in the shaded boxes at the right part of the figure. The arrows represent "based on" relations. Taking into consideration the aforementioned criteria, this work will examine the most recent PL methods of fig. 2.3. These methods have also found broad acceptance in the software industry and are well documented. The methods are briefly examined from the perspective of the identified issues of feature-architecture mapping, feature-level variability and product instantiation. A comparison of the various approaches regarding each of the issues is made and the most representative methods are selected for more detailed examination.

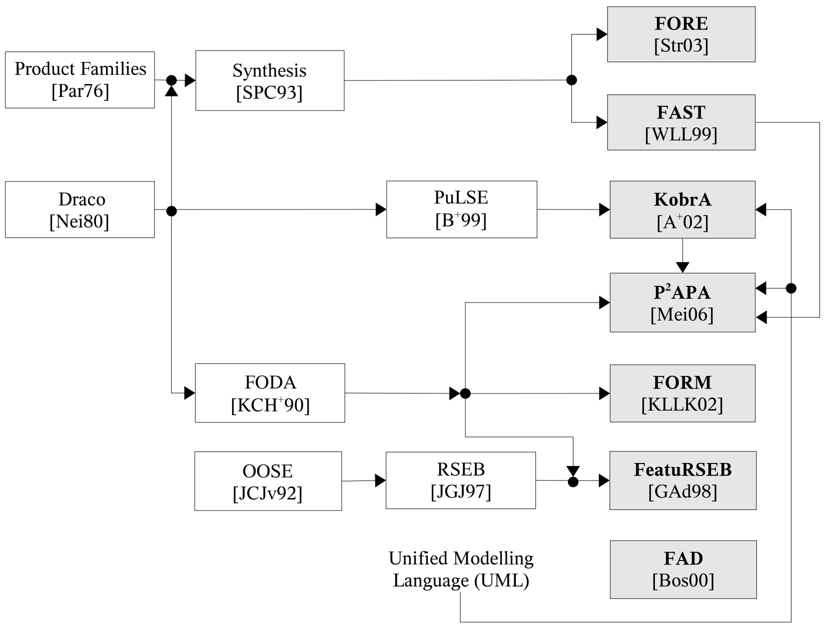


Figure 2.3: Historical evolution of PL methods

FeatuRSEB The *featured* RSEB method [GAd98] is based on a combination of the Feature Oriented Domain Analysis (FODA) method [KCH+90] and the Reuse-Driven Software Engineering Business (RSEB) method [JGJ97]. FeatuRSEB combines the use-case models of RSEB with the feature modeling concepts of FODA. FeatuRSEB architectural components are derived from use-cases. The mapping between features and components is achieved through *traces*. Feature-level variability is achieved through generalization and specialization techniques and design patterns. FeatuRSEB takes a manual approach to application engineering. PL products are individually developed through the extension and combination of the PL architectural components.

KobrA The KobrA method [A+02] is a descendant of the Product-Line Software Engineering Methodology (PuLSE) [B+99]. KobrA concretizes the development phases of PuLSE and integrates them with UML. KobrA components are also derived through use-case modeling techniques. Mapping between features and the architecture is supported through a *decision model*. The latter is similar to the concept of traces. The incorporation of variability mechanisms in KobrA products is done similar to FeatuRSEB. The KobrA product instantiation is based on object-oriented frameworks.

FAD Functionality based Architectural Design (FAD) [Bos00] is a methodology developed by the RISE group at the university of Karlskrona/Ronneby Sweden. The method has been applied to a variety of domains. FAD components are, among others, instantiations of *archetypes*. Archetypes are the core abstractions based on which the system is structured. FAD provides no mapping mechanism between features and the architecture. The feature-level variability mechanisms of FAD are similar to those of FeatuRSEB and KobrA. FAD, like KobrA, employs object-oriented frameworks for product instantiation.

FORM The Feature-Oriented Reuse Method (FORM) [KLD02] is a concretization of the FODA method. It provides a more detailed description of the domain engineering processes and incorporates a marketing and product plan aspect into the method. FORM components are derived similar to those of FeatuRSEB. FORM provides no extra mechanism for a mapping between features and components. FORM's variability mechanisms and product instantiation do not differ from those of FeatuRSEB.

FAST Family-Oriented Abstraction, Specification and Translation (FAST) [WL99] is based on the Synthesis method [SPC93]. Nonetheless, its processes and artifacts are similar to those of PuLSE. FAST provides a very solid domain analysis methodology, namely, the *Commonality Analysis*. Unfortunately, the method's processes are described on a rather abstract level, with no concrete reference to an implementation technology. Because of this fact, FAST gives no direct answers to the issues examined in this work.

FORE Family-Oriented Requirements Engineering (FORE) [Str03] is also based on the Synthesis method [SPC93]. It is a methodology designed particularly for modeling of large systems through the use of FMs. System requirements are captured by means of an extended feature notation and are also represented in form of a machine readable data model. FORE provides automatic tools for the validation/selection of PL products. The mapping to the architecture in FORE remains vague, since the focus of the method lays on product instantiation. Nonetheless, some parts of the new extended notation are used in this work (fig. 2.2).

P^{2}APA The Product-driven Pattern-oriented Agile Product-Line development Approach [Mei06] is influenced from the FAST, KobrA and FODA methods, as well as the UML. P^{2}APA's focus lays on the evolutionary, versatile development of PLs suitable for companies with limited resources. Components are conceptualized in P^{2}APA through the definition of PL-Archetypes, which are then concretized with methodologies similar to those of KobrA. Non-functional requirements are addressed through the use of architectural and design patterns. P^{2}APA provides an implicit mapping between features and the architectural components through the use of feature graphs. The issue of feature-level variability is handled on an abstract basis through the introduction of the new concepts of degree of freedom and orthogonal documentation. The product instantiation of P^{2}APA is similar to the one of KobrA.

Based on the brief description of each method, it is clear that FeatuRSEB provides a solid approach to feature-architecture mapping. FORM's processes are similar to those of FeatuRSEB. Therefore, by addressing FeatuRSEB, one also handles the issues of FORM. KobrA's feature-architecture mapping approach is similar to that of FeatuRSEB. Additionally, KobrA's approach to application engineering resembles the one of FAD, since both methods make use of object-oriented frameworks. Furthermore, FAD archetypes present an interesting approach to component development. Thus, by addressing FeatuRSEB and FAD's component development process and object-oriented frameworks, one also handles the main parts of the KobrA method. As mentioned in the descriptions of FAST and FORE, the lack of concretization for the FAST processes and the focus on product instantiation of FORE provide no ideal ground for further analysis. The P^{2}APA method provides a solid abstract approach to feature level variability, but similarly to FAST, no concrete implementation. Additionally, its lack of an explicit feature-architecture mapping, as well as the use of a KobrA-similar product instantiation approach, allow the delegation of its handling within the context of FAD. Finally, all aforementioned methods follow the same approach to feature-level variability. This issue can therefore also be addressed in the context of FeatuRSEB.

Concluding the above discussion, FeatuRSEB and FAD are mature PL methods that provide a representative picture of the approaches taken today for the resolution of the identified issues. Based on this, sections 2.3 and 2.4 will give insight into the processes of these methods with respect to feature-architecture mapping, feature-level variability and product instantiation. More precisely, feature-architecture mapping and feature-level variability will be examined in the context of FeatuRSEB in section 2.3. Furthermore, feature-architecture mapping and product instantiation will be examined within the context of FAD in section 2.4. Finally, section 2.5 will evaluate the approaches taken by the generative programming techniques for the resolution of the aforementioned issues.

2.3 Featured Reuse-Driven Software Engineering Business

The Featured Reuse-Driven Software Engineering Business (FeatuRSEB) method is a concrete PL method, providing state of the art solutions to the issues on feature-architecture mapping and feature-level variability. With respect to feature-architecture mapping, section 2.3.2 will examine and evaluate the sources of the FeatuRSEB components, as well as the mapping between PL features and components. Feature-level variability will be explored from the perspective of the various variability mechanisms and the way they are applied for implementing variability on a PL feature in section 2.3.3.

2.3.1 Overview

FeatuRSEB is a result of the integration of FODA's [KCH$^+$90] domain analysis process with the processes of RSEB [JGJ97]. The developers of FeatuRSEB placed the FODA FM at a central position in relation to the other models of RSEB. The FM in FeatuRSEB plays a unifying role, tying all RSEB models together.

Fig. 2.4 shows the processes, workers and products of FeatuRSEB. FeatuRSEB is divided into **Application Family Engineering** (AFE), **Component System Engineering** (CSE) and **Application System Engineering** (ASE). All processes exchange information and proceed concurrently.

AFE is initiated by the lead architect who constructs the PL architecture. The latter is always a layered architecture, consisting of a family use case model and a family design model. From this layered architecture an initial FM is developed by the domain analysts. The architecture also defines the interfaces between the CSE and ASE processes, which are at that point concurrently started.

Through feedback from the customers and end users of the PL, the application use case and sub-system engineers construct an application use case and design model for the various applications covered by the PL. The component use case and sub-system engineers take advantage of the commonalities and variabilities in the PL

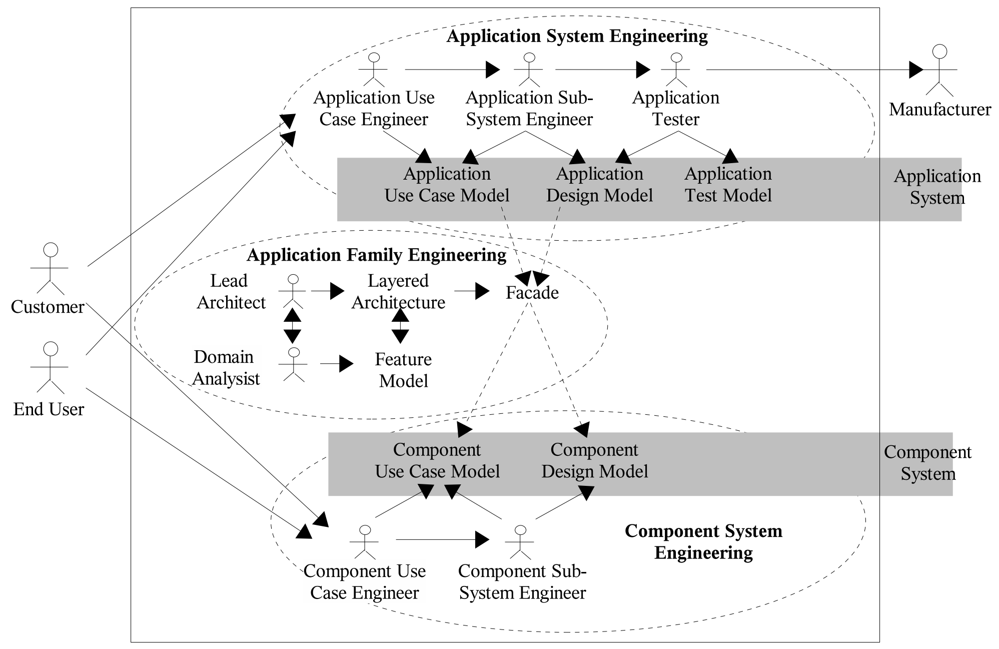


Figure 2.4: An overview of the FeatuRSEB processes

applications and construct component use case models and the respective component design models. Through facades defined in the AFE process the application engineers import and reuse use cases or design objects from the component system of the CSE process. The application tester tests each application and gives feedback to the application engineers. Possible additions or corrections are then performed that may influence all artifacts of the FeatuRSEB processes.

Finally, a manufacturer receives an application system, composed of the artifacts shown in the upper shaded part of fig. 2.4, namely, an application use case model, a design model with implementation and a test model. The manufacturers are responsible for making customizations to meet local needs and for distributing the application. These activities are outside the scope of FeatuRSEB.

FeatuRSEB's layered architecture and its relation to the FeatuRSEB processes is shown in fig. 2.5. AFE is concerned with the construction of the architecture as a whole and defines the interfaces between the various layers, e.g. between the applications and business-specific component systems. In CSE the component systems in the lower layers of the architecture are developed through various processes for each component system, as shown by the multiple eclipse symbols on the lower left part of the figure. Similarly in ASE, the various applications of the PL are developed in distinct processes.

The application systems offer a coherent set of use cases to some end users. The

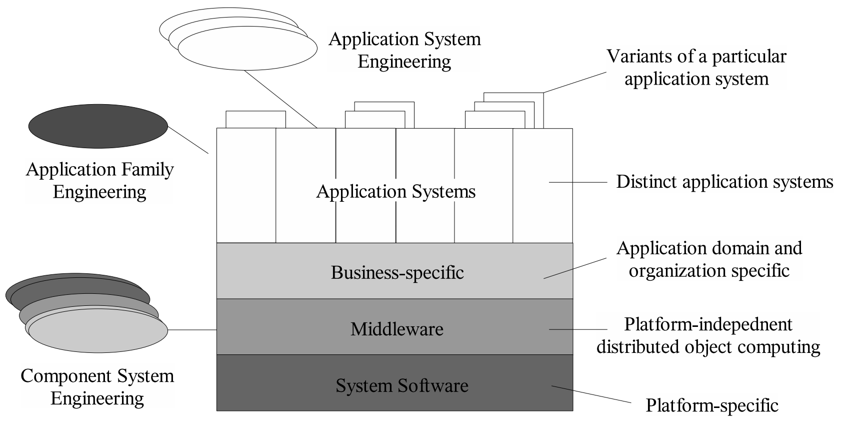

Application System Engineering
Variants of a particular application system
Application Family Engineering
Application Systems
Distinct application systems
Business-specific
Application domain and organization specific
Middleware
Platform-indepednent distributed object computing
Component System Engineering
System Software
Platform-specific

Figure 2.5: A typical FeatuRSEB layered architecture and its relation to the FeatuRSEB processes

business-specific layer contains a number of component systems specific to the type of business. The middleware layer offers component systems for utility classes and platform-independent services, e.g. distributed object computing in heterogeneous environments. Finally, the system software layer contains the software for the actual infrastructure, such as operating systems, interfaces to specific hardware, etc..

The relation between application and component systems, as well as the internal structure of a component system are shown in fig. 2.6. The figure shows application systems for a banking software PL. The application systems import objects from the various component systems through the facades provided by the component systems. For instance, the application system on the upper right part of the figure imports through the facade of the `Account Management` component system the `Account`' and `Transaction`' design objects. As shown also from the figure, the internal structure of a component system in FeatuRSEB is composed of a use case, a design and an implementation model in separate packages of the respective type.

2.3.2 Component Sources & Mapping

In order to enhance stability, the PL architecture must illustrate high cohesion and low coupling on the feature level. The latter imply a strong mapping between features and the architecture. This section will examine and evaluate the different factors that influence feature-architecture mapping, namely, the source from which components are derived and the mechanism used for the actual mapping between features and the architectural components.

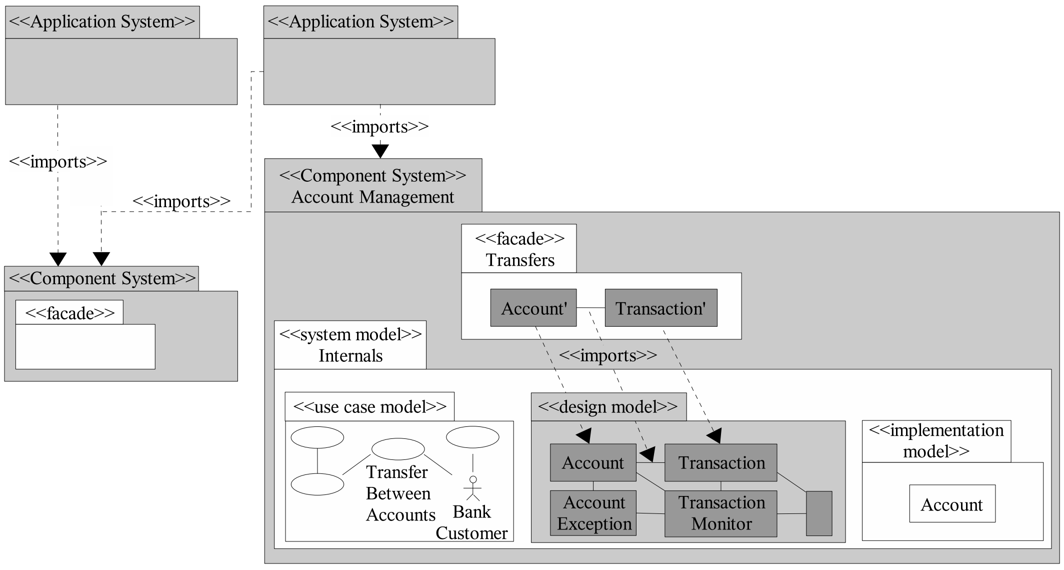


Figure 2.6: Relations and internals of application and component systems

Component Sources

Feature-architecture mapping depends on the sources used for the derivation of the components of the PL architecture. The component sources have a direct influence on the responsibilities of the architectural components. For instance, if one uses features as a source for the architectural components, it is most likely that the derived components implement the features partially or even as a whole. The use of the solution domain for the derivation of components, e.g. the *Blackboard* architectural style [Bus96], increases the likelihood of deriving components having little or no relation to the PL features.

FeatuRSEB's software architecture is constructed during the so called *robustness analysis*. During this process, a high-level, static structure, which shows types, their grouping and relationships with other types, is created. This is captured in the FeatuRSEB's *analysis model*. This is a model of the system design at a high level, which ignores the specific low-level details of the target implementation environment, i.e. the PL architecture.

The entities of the analysis model represent the components of the architecture. They are later defined in more detail in the design model and are eventually implemented. The analysis entities are derived from the use cases. The FeatuRSEB analyst searches through the description of the requirements and use cases, looking for the elements that can adhere to the FeatuRSEB analysis types. The analysis types of FeatuRSEB are:

Entity types are long-lived objects in the system. They outlast the use case instances in which they participate. They are often used to model business objects that represent "things", such as accounts and loans, dealt with in many use cases. Entity types are thus generic to many use cases.

Boundary types handle the communication between the system and its surroundings. They actually transform events and objects from the system's representation to a representation suitable for its surrounding and vice versa. They constitute the presentation-dependent part of the system, leaving the other types surrounding-independent.

Control types perform use-case-specific behavior. They often control and coordinate other objects. They offer behavior that does not belong to an entity or boundary type.

Fig. 2.7 shows the various types across the FeatuRSEB's analysis dimensions. Entity types model information in the system that should be held for a long time. All behavior naturally coupled to this information should also be placed in the entity object. A boundary type models behavior and information that is dependent on the interface of the system. The control type models functionality that is not naturally tied to any other type. Typically, this is behavior consisting of operations on several different entity objects, doing some computation and then returning the result to a boundary object.

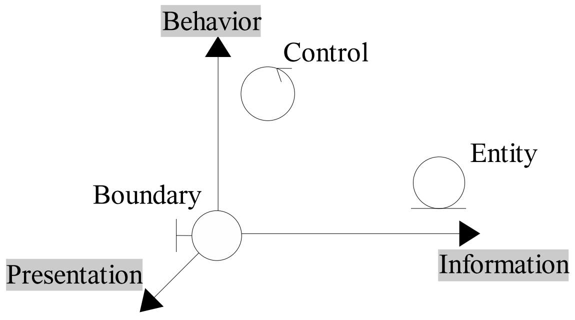


Figure 2.7: The dimensions and types of the FeatuRSEB analysis model

An example from the book on RSEB [JGJ97] is shown in fig. 2.8. The analyst has used the `Withdraw Money` use case to derive the `Cashier Interface`, `Dispenser`, `Withdrawal` and `Account` components. The original text for the `Withdraw Money` use case is given below:

Withdraw Money Use Case Description: Analysis model

The **Bank Customer** chooses to withdraw money. The `Cashier Interface` first asks the **Bank Customer** to identify himself or herself.

If the identification is successful, the `Cashier Interface` asks the **Bank Customer** to choose how much to withdraw and from which `Account`. The `Cashier Interface` orders the **Withdrawal** object to confirm that the **Bank Customer** has the right to withdraw that amount from the `Account`. The **Withdrawal** object validates the request.

If the **Bank Customer** can withdraw that amount, the **Withdrawal** object asks the `Dispenser` to dispense the amount and deducts the amount from the `Account`.

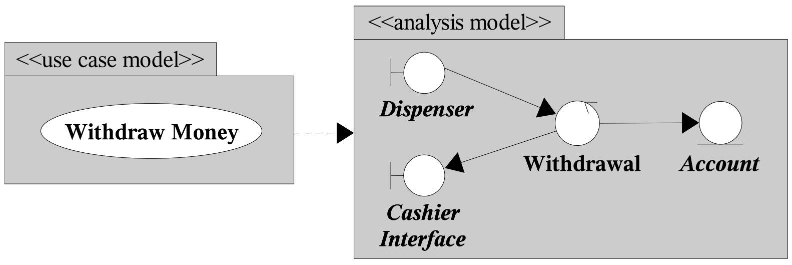


Figure 2.8: The Withdraw Money use case and the corresponding partial analysis model

Evaluation of Component Sources

The utilization of use cases for the derivation of components in PLs does not inherently enhance feature-architecture mapping. Features are derived from a variety of sources, e.g. existing systems, literature, domain experts and requirement specifications. FMs provide the "which" of the domain, i.e. which functionality can be selected when engineering new systems in the domain. Thus, FMs capture the commonality and variability of the domain. Use cases models provide the "what" of the domain, i.e. a complete description of what systems in the domain should do. This fundamental difference prevents the derivation of components having a strong mapping to features.

FeatuRSEB derives components based on use cases and orders them within the three dimensions of presentation, information and behavior respectively deriving boundary, entity and control types of components (fig. 2.7). One can say that components of boundary type tend to support feature-architecture mapping. This is because features are primarily visible from the customer point of view. Boundary types are visible to the actors of the use cases. Although there exists no exact overlap of the concerns of customers and actors, boundary components may in some cases provide components with a strong mapping to features.

Entity components model under certain circumstances objects of the problem domain. In these cases, entity components do have a strong mapping to features. In any other case, entity objects represent objects based on an "artificial" domain, which in fact promotes feature scattering and tangling.

Control components capture behavior not belonging to the previous two types. This fact alone is an indicator of functionality coming from the solution domain. The solution domain is not visible to the customer, rather to the software engineer. Thus, control components do not naturally support feature-architecture mapping. Nevertheless, control type components may be required for the solution of an implementation or architectural problem. In some cases they also serve the changeability and stability of the PL architecture.

An example supporting the above argumentation is taken from the book on RSEB [JGJ97]. Fig. 2.9 shows the relation between the **Withdraw** and **Deposit** features, their respective use cases and architectural components.

In this example it is assumed that there exists a one to one relation between features and use cases. This is not always the case, rather the exception, but it will be used for simplicity reasons. As shown in the figure, the **Withdraw** feature is expressed in the `Withdraw Money` use case. The latter is implemented in the `Dispenser`, `Cashier Interface`, `Withdrawal` and `Account` components, indicated by the respective shading of the analysis types. Respectively, the **Deposit** feature is expressed with the `Deposit Money` use case, which is implemented in the `Money Receptor`, `Cashier Interface`, `Deposit` and `Account` analysis types.

It is obvious that both features are scattered and tangled throughout the system. The **Withdraw** feature is scattered throughout all white-colored components and the **Deposit** feature throughout all shaded components. Additionally, both features are tangled within the `Cashier Interface` and `Account` components.

As shown from this example, the use of use cases and the FeatuRSEB's analysis dimensions have not prevented the effect of scattering and tangling. The direct use of features for the derivation of components could lead to the architecture of fig. 2.10. In this architecture, one component for each feature is derived, containing the respective sub-components, as in the FeatuRSEB architecture. For example,

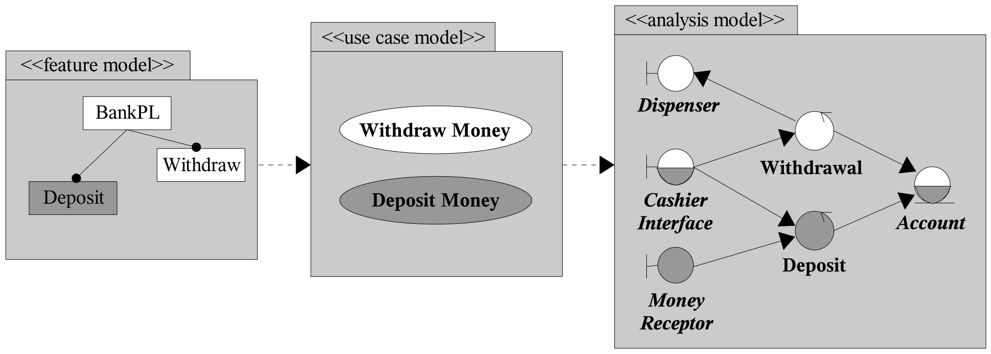


Figure 2.9: Scattering and tangling of the **Withdraw** and **Deposit** features

the `Withdraw` component is derived from the **Withdraw** feature and has the sub-components `Dispenser`, `Withdraw` and `Cashier Interface`.

The main difference from the FeatuRSEB architecture (fig. 2.9) is that the `Cashier Interface` sub-components now implement only the specific functionality of the respective feature. A possible implementation of this could be the utilization of a common interface platform, providing a plug-in functionality for the addition of menus and dialogs. This would then allow the `Cashier Interface` sub-components to add their interface controls on the common system interface and directly handle the events triggered on these controls. This architectural division resolves feature scattering and tangling on the system's interface.

Nonetheless, the presented architecture does not resolve feature scattering and tangling in the `Account` component. This is an example of how a FeatuRSEB control type may serve the system changeability and stability. The functionality provided by the `Account` component is needed from various other components of the system. Therefore, the `Account` component represents a strong concern from the architectural perspective and needs to be implemented as a separate component.

The architecture of fig. 2.10 has provided a compromise between features and architectural concerns. In the FeatuRSEB architecture (fig. 2.9), the tangling of the interface implementations of the **Withdraw** and **Deposit** features does not guaranty that a change in the interface of one feature does not impose changes on the interface of the other feature. This depends alone on the implementation of the `Cashier Interface` component. In the architecture of fig. 2.10, the developers may use e.g. the capabilities provided by the common interface platform, to incorporate changes on the feature interfaces with minimal influence on other features.

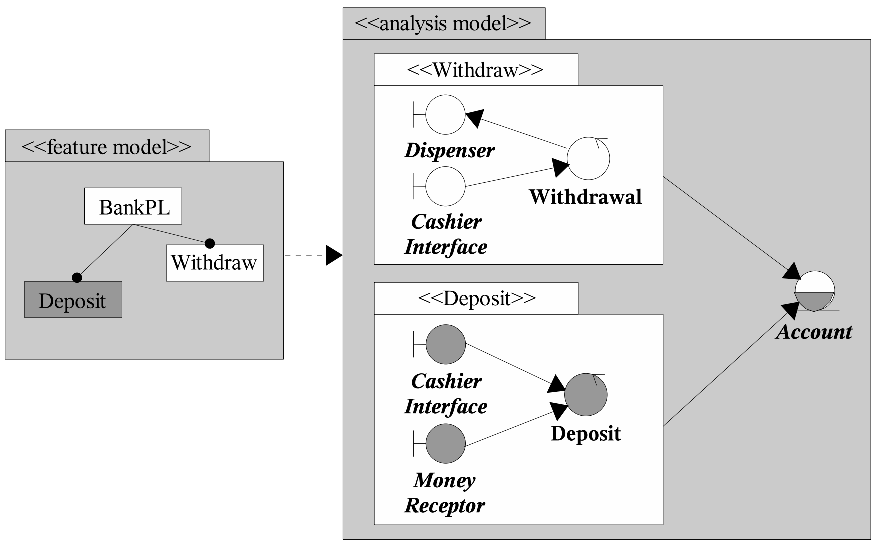
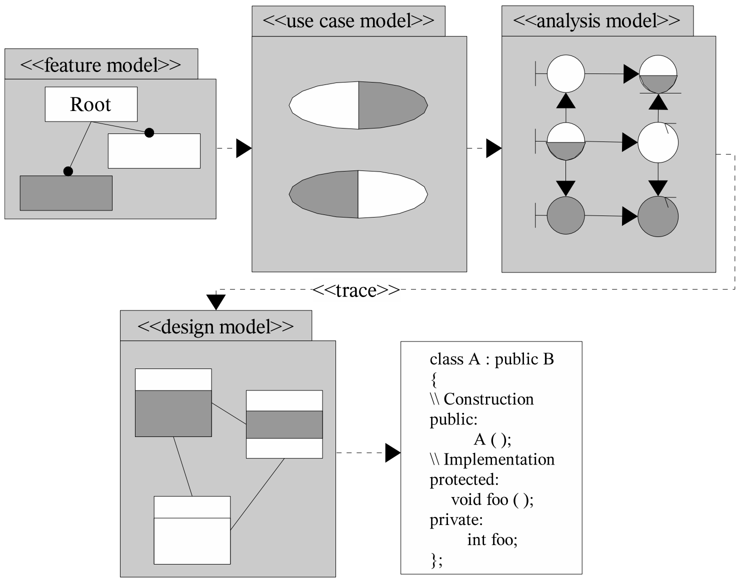


Figure 2.10: An architecture based on the direct derivation from features

Mapping Mechanism

Another crucial point for a strong mapping between features and the architecture is how efficiently do derived components map to the PL features. For instance, when derived components have a weak mapping to features, the implementation of a mapping mechanism can prove to be extremely inefficient. This is due to the fact that the feature implementation is spread throughout the components of the software architecture. In the case where feature scattering and tangling is limited, the actual mapping between features and their implementation is significantly simplified and thus more efficient.

The different models of FeatuRSEB and the elements defined in the different models are connected with each other by ≪*trace*≫ dependencies. Fig. 2.11 shows the traces between the various models of FeatuRSEB. Features are mapped to the respective parts of use cases. The use cases are then mapped to the components of the analysis model. The components of the analysis model are mapped to the concrete design model entities. Finally, the design model entities are mapped to the system implementation. FeatuRSEB traces are shown by the dashed arrows. In order to reduce the complexity of the figure, the white and shaded parts of the figure indicate the traces between the elements of the different models.



```
class A : public B
{
\\ Construction
public:
        A ( );
\\ Implementation
protected:
    void foo ( );
private:
        int foo;
};
```

Figure 2.11: Feature-architecture mapping in FeatuRSEB

Evaluation of the Mapping Mechanism

Because of feature scattering and tangling, the number of traces needed to map a feature to the architecture and eventually to its implementation is extremely large. This becomes even more evident in software intensive PLs, consisting of hundreds of KLOC (1000 Lines Of Code). High effort is required for the creation and maintenance of the traces, even for small-sized PLs. The reduction of the effort for this task could of course be supported from special purpose tools. Works in this field can be found in [MR02] and [JZ05]. Respectively, the work in [MR02] identify the problems denoted earlier and stresses the importance of traceability in the context of PLs. [JZ05] attempt to provide a rule-based approach for the automatic generation of traceability links. [MR02]. Nonetheless, tool support does not reduce the inherit complexity of the task.

The use of traces in FeatuRSEB is highly inefficient for the achievement of a strong mapping to the architecture. Traces are used to compensate for the effect of feature scattering and tangling, which is caused during the architecture development. Traces treat the symptoms of feature scattering and tangling, while failing to address the root of the problem.

2.3.3 Feature-Level Variability

One of the two preconditions identified for the success of a PL approach is the efficient instantiation of PL products. Since PL products are defined as a set of features, ideally, a product should be automatically generated based on these features. For this reason, the PL architecture should enable variability on the feature level. More precisely, it should at least enable the inclusion of a feature in a PL product at compile-time. For PLs having higher flexibility requirements, the PL architecture should also enable adding or removing features of a product on start-up-time or even runtime and dynamically switching between features at runtime.

Variability in Software Product Lines

In a PL, the common software architecture developed during PL engineering needs to be bound to a set of variants for the instantiation of a PL product. These variants must implement the desired set of features that the product should possess, as well as the needed variability for these features.

In order to enable the aforementioned variability on the feature level, a number of variability mechanisms must be applied on the set of variants in the PL architecture. Which variability mechanism to use for each of the above cases depends on two factors: the point in time in which the variant is bound to the architecture, referred to as binding time and the type of the variation at hand [GBS01], [BB01].

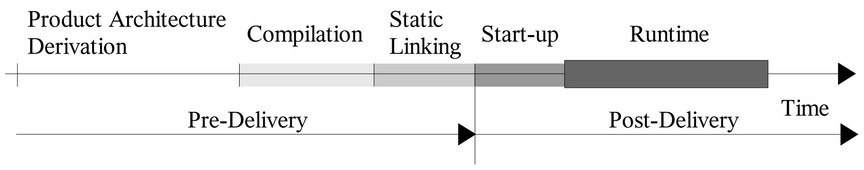


Figure 2.12: Binding times of PL product variants

The binding times of a variant are shown in fig. 2.12 and are explained below:

- **Pre-Delivery**
  - *Product Architecture Derivation* The binding of the open variation points of the PL architecture is done early on the level of architectural design. Typically, configuration management tools are involved in this process.
  - *Compilation* The finalization of the source code of the PL product is done during the compilation process. This includes pruning the code according

  to compiler directives in the source code, but also extending the code to superimpose additional behavior (e.g. macros and aspects).

  - *Static Linking* Variants can be added to a PL product through linking of library files. Linking is performed at this stage irrevocably right after compilation.

- **Post-Delivery**

  - *Start-up* Some decisions for the inclusion of a variant in the running product must be taken at the customer's site. These decisions can be made just before the system starts, i.e. at start-up-time. This can be achieved through the use of configuration files dictating the system which modules to load. For example, dictating which dynamic libraries should be linked to the system.

  - *Runtime* An application is rendered interactively at runtime. For example, the PL product may have the ability to communicate with the outside world by using different communication protocols in parallel. New variants can be added to the system through the use of object-oriented techniques. A number of design patterns can be used for this purpose e.g. the Strategy design pattern [BJM$^+$95].

The types of variability that may occur on a variation point in the architecture are:

- **Optional** A variant is either included in the architecture or not. When the optional variant is used by other parts of the architecture, variability mechanisms must be applied to deal with the case when the variant has not been included in the architecture.

- **Alternative** One variant from a set of alternative variants can be included into the architecture. In this case the architecture provides a placeholder in which one of several alternatives can be inserted. Other parts of the architecture that depend on this variation point must be able to communicate with the different alternative variants.

- **Set of Alternatives** Multiple instances of different alternative variants can be included into the architecture. In this case, the variability mechanism must support several instances of alternate variants running in parallel in the system. This type of variability occurs only during runtime. The variability mechanisms for this type of variability are design patterns, e.g. Strategy and condition on variable.

Variability Mechanisms		
	Type of Variability	
Binding Time	*Optional*	*Alternative*
Product Architecture Derivation	- "Null" Component	- Configuration Management - ADLs
Compilation	- Condition on Constant	- Code Fragment Superimposition - Condition on Constant
Static Linking	- Explicit Linking	- Explicit Linking
Start-up	- Dynamic Linking	- Dynamic Linking
Runtime	- Condition on Variable	- Infrastructure-Centered Architecture - Condition on Variable

Table 2.1: Variability Mechanisms in relation to binding time and the type of variability

Depending on the binding time and the type of variability to be supported, numerous variability mechanisms exist. For the discussion of the feature-level variability issue, a representative set of variability mechanisms for the possible combinations suffices. Table 2.1 shows the set of variability mechanisms for optional and alternative types of variability. Note that the variability mechanisms for the set of alternatives type of variability refer only to the runtime binding time and are listed above. The following sections will explore the variability mechanisms in more detail.

Optional Type of Variability

Optional variants of PL products must be bound during the product's architecture derivation. A common variability mechanism is *"Null" Component.* A dummy component is developed supporting the same interface as the functional optional component. The dummy component returns dummy values to other parts of the architecture needing the optional variant. This mechanism ensures the proper operation of the system even in the absence of the optional component.

During compilation, the dummy or the functional component implementation can be selected. This can be achieved through the use of pre-processor directives, e.g. C++ `ifdef` statements. These opt out the code related to the dummy or functional component respectively.

In static linking, the dummy or functional components can be implemented as library files, which are properly included in the PL product through linker directives. The same principle can be applied during start-up, although this time the components must be built into dynamic libraries, e.g. Windows DLLs.

During runtime, a condition on a variable can be used to select between the dummy and functional implementation of the variant. These are the common decision structures in programming languages, e.g. if-statements or case structures.

Alternative Type of Variability

Early binding of exactly one variant from a set of alternatives can be achieved through configuration management tools or Architecture Description Languages (ADLs). The former approach utilizes a tool to select one variant from a set of alternatives for inclusion in the PL product. The latter uses an ADL to produce a definition for the selected variant, whose implementation is then generated from the ADL platform.

This type of variability can also be dealt with through code fragment superimposition during compilation. The software architecture is developed in a generic way and product-specific concerns are superimposed on the completed source code. There exist a number of generative programming techniques, e.g. Aspect-Oriented Programming [Kic97] or the Hyperspace approach [OT01]. The details and evaluation of these techniques are discussed in section 2.5.

In order to be able to select a variant from a set of alternatives at runtime, an infrastructure-centered architecture can be used. This approach makes the connections between components in the architecture a first class entity. The components are no longer connected directly to each other, rather to the infrastructure. The latter is then responsible for matching the required interface of a component with a provided interface of one or more components. The infrastructure can be an existing standard, e.g. COM or CORBA, or it can be an in-house developed standard. It may also be a scripting language that "glues" components together [Ous98].

The condition on constant, condition on variable and the explicit and dynamic linking mechanisms can also be used as described for the optional type of variability.

Set of Alternatives Type of Variability

The last type of variability refers to variants running in parallel in the system. For this reason, it is only meaningful to discuss this type of variability after the product has been instantiated and is already running. A number of design patterns [BJM$^+$95] exist that allow the runtime selection between numerous alternatives running in parallel. Basically, they implement several component implementations adhering to the same interface and make these component implementations tangible entities in the system architecture. An example is the Strategy design pattern that allows having several implementations present simultaneously. The patterns Abstract Factory and Builder provide ways of making sure the correct implementation gets the data.

The use of the condition on variable variability mechanisms is applied in the same way as described in the optional type of variability.

Evaluation of Feature-Level Variability

The presented set of variability mechanisms is used by the state of the art PL methods to enable variability on a PL feature for the instantiation of PL products. The effect of feature scattering and tangling though does not allow the efficient application of these variability mechanisms. The following sections will evaluate the efficiency with which the various variability mechanisms can be applied on the feature level in the presence of feature scattering and tangling.

"Null" Component

It is first assumed that a feature's implementation is only scattered throughout a number of components and that these components implement no other features, i.e. there is no tangling. In order to apply the "null" component variability mechanism, for each of the aforementioned components a "null" component must be created. When the feature is selected, the functional versions of the components must be included into the architecture and when the feature is not selected, all dummy versions of the components must be included into the architecture.

In the presence of both scattering and tangling there exist two possibilities for the application of this variability mechanism. The first one is the extraction of the feature-specific functionality into new components illustrating no tangling. Then the previously mentioned scenario can be applied.

Another possibility is the isolation of the feature-specific functionality within the original component and the creation of feature-specific interfaces for that functionality. Then a "null" component can be developed that replaces the entire original component, providing dummy implementations for the feature-specific interfaces.

The effort to apply this variability mechanism in the presence only of scattering can be characterized as medium. The effort is directly dependent on the number of components within which the feature is implemented and the number of interfaces they support. In the presence of both scattering and tangling, the effort needed increases dramatically and is unpredictable. In this case, both aforementioned possibilities require either the extraction or the isolation of the feature-specific functionality in the original component. If this is feasible or not depends on the intensity of the tangling, i.e. how many features are tangled within the original component and in what ways. It is very likely that high effort is needed for a developer to understand these implications and resolve them. Nonetheless, the effort needed depends on the case at hand and cannot be foreseen.

Condition on Constant & Condition on Variable

The condition on constant and condition on variable mechanisms are rather similar with respect to their application and will be examined together. Both variability mechanisms work on the code level. This makes them very flexible to apply in the case of feature scattering and tangling. More precisely, each part of a feature's implementation can be surrounded either from a compiler directive (condition on constant) or an if-statement (condition on variable) and can be varied at compile or runtime respectively. Since these variability mechanisms are so fine grained and can be applied on the code level, a feature's implementation can be isolated almost in every case.

Unfortunately, this variability mechanisms require high effort to apply and deteriorate the system's conceptual integrity. For large scale systems, like in the case of PLs, the number of source code lines for a single feature can be significantly high. Finding and isolating a feature's source code can be consequently extremely time consuming. Even in the case where such a task is completed, the quality of the source code would have considerably deteriorated. This is due to the fact that extra constructs must be added, e.g. ifdefs, if-else-statements, case structures, etc., that jeopardize the system's understandability.

Explicit & Dynamic Linking

The application of explicit and dynamic linking as variability mechanisms requires that a feature is implemented in distinct library files. The latter should illustrate no tangling, i.e. no other features' implementations must be included in the library files.

As in the case of the "null" component variability mechanism, the effort needed for such division is unpredictable. A feature whose implementation is scattered and intermingled throughout a number of components may prove very difficult or even impossible to isolate.

Configuration Management & ADLs

Using configuration management for enabling variability on a feature is certainly very efficient, after it has been implemented. A tool is then able to receive e.g. the name of the feature and it can include or exclude the components implementing the feature. Like in the previous cases of the linking and "null" component variability mechanisms, this variability mechanism assumes a strong mapping of the feature to the architecture, i.e. no tangling should exist. The argumentation is thus identical to the previous cases.

The use of an Architecture Description Language (ADL) for achieving variability on an alternative feature is comparably efficient to configuration management. The definitions of the various components realizing the feature are given as input to the ADL tool, which then generates them.

The issues that arise with this variability mechanism are on the one hand that it cannot be applied to pre-existing features and their components and on the other hand, the actual implementation of the ADL. The former issue sets as a precondition that the entire PL architecture should have been developed with the ADL. This excludes the use of this variability mechanism with existing PLs, which are not based on an ADL. The latter issue has to do with the effort required for the definition of the ADL constructs and their implementation and maintenance.

Infrastructure Centered Architecture

This variability mechanism requires an infrastructure based architecture that transforms the component connections to first class entities. This precondition can be restrictive, e.g. in the case of systems with high performance requirements or architectural requirements that exclude an architectural structure such as COM or CORBA.

Using a scripting language to achieve a similar effect does not have such a large impact on the software architecture. This mechanism is very flexible and may prove to be sufficient for a variety of domains with less strict performance requirements. Nonetheless, using a scripting language for the implementation of variability on a feature may have a negative impact on the system's conceptual integrity. The "glue" code introduced to decide if the feature's code is activated or not, adds extra inhomogeneous entities in the PL architecture, thus diminishing its understandability and maintainability.

Design Patterns

The use of design patterns for the implementation of variability on a feature can be discussed in the context of the Strategy design pattern [BJM$^+$95], shown in fig. 2.13. The Strategy design pattern enables the selection of one variant from a set of alternatives running in parallel.

A **Context** object provides an interface to other client objects in the system, allowing runtime variability between different strategies (variants). The abstract ***Strategy*** class defines an `AlgorithmInterface()` method. The latter is implemented in each of the **ConcreteStrategy** objects for every alternative variant. The **Context** object is set to use one of the concrete strategies through its `ContextInterface()`. The **Context** object will now use one of the given variants, depending on which concrete Strategy has been set.

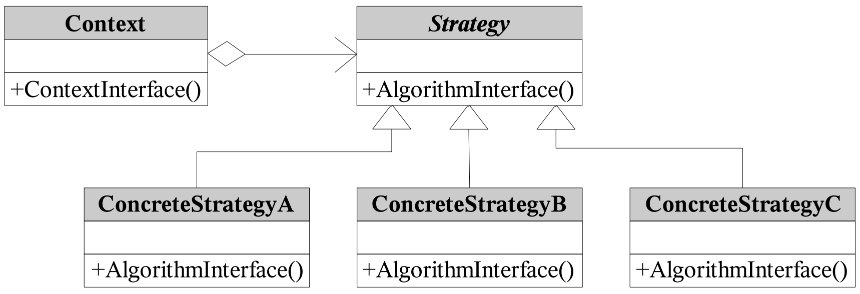


Figure 2.13: UML diagram of the Strategy design pattern

It can be seen from this example that design patterns in general operate on the detailed design level. In order to use the pattern, an alternate feature's implementation must be simple enough, so as to allow its complete encapsulation within a simple interface. If this is not the case, then the design pattern must be applied throughout the software system where the parts of the feature's implementation are scattered. This introduces numerous inhomogeneous entities. For instance, in the case of the Strategy pattern this would cause the addition of an abstract ***Strategy*** class, along with the **ConcreteStrategy** classes for each different part of the feature's implementation throughout the system.

From the above discussion it can be concluded, that the use of design patterns for the implementation of variability on a feature is quite efficient for simple, low level features. In the case of feature scattering and tangling, design patterns tend to introduce architectural entities having little or no relation to their environment.

This has a negative impact on the system's conceptual integrity.

Furthermore, as in the previous variability mechanisms, the effort needed to "untangle" a feature for the direct application of a design pattern is unpredictable.

2.3.4 Conclusions

This section on FeatuRSEB has provided an overview of the method and an evaluation of its feature-architecture mapping and efficiency of variability mechanisms. The feature-architecture mapping has been examined based on the component sources and the actual mapping mechanism between features and the architecture. The efficiency of the variability mechanisms has been evaluated based on a representative set of the available variability mechanism.

The feature-architecture mapping in FeatuRSEB can be generally characterized as insufficient, although it does illustrate a few positive aspects. More precisely, the utilization of use cases for component derivation is rather inappropriate for a strong mapping between features and the architecture. The use of the boundary, entity and control analysis types for the derivation of the architectural components has both positive and negative effects on feature-architecture mapping. Boundary and entity types support feature-architecture mapping when representing concerns closer to the costumer perspective, while control types do not normally support feature-architecture mapping, although there may exist situations where their presence supports the system's maintainability and flexibility. Furthermore, the actual mapping mechanism used in FeatuRSEB, i.e. ≪*trace*≫ dependencies, is very inefficient for large scale PLs. A large number of traces can very quickly become unmanageable. Tool support can help in this respect, but it does not solve the actual problem of feature scattering and tangling.

The variability mechanisms have been presented and evaluated in this section of FeatuRSEB, but they are also common to the rest of the state of the art PL methods. It has been shown that in certain cases the available variability mechanisms can be efficiently applied on the feature level in the absence of feature tangling. Examples are the "null" component, linking and configuration management. Other variability mechanisms set significant preconditions, e.g. the use of CORBA or COM, as an infrastructure for the PL or the use of an ADL. Yet other variability mechanisms can be efficient only in the case of small features, e.g. condition on constant, condition on variable and design patterns.

All variability mechanisms may introduce a large number of inhomogeneous entities in the PL architecture, jeopardizing the system's conceptual integrity [Bro95] and thus its understandability and maintainability. Finally, all variability mechanisms may cause unpredictable amounts of effort for their application depending on the degree of feature tangling.

Goal of this work is to achieve a stronger mapping between features and the architecture and to efficiently apply the available variability mechanisms.

2.4 Functionality-based Architectural Design

The Functionality-based Architectural Design (FAD) method [Bos00] will be examined from the perspective of feature-architecture mapping and product instantiation. FAD provides no explicit mechanism for the mapping between features and the architectural components. Nonetheless, FAD components illustrate high maintainability and promote the system's conceptual integrity [Bos00]. Section 2.4.4 will discuss the pros and cons of the FAD component development process from the point of view of feature-architecture mapping. Goal of this work is to enable an efficient generative approach to product instantiation. FAD's product instantiation takes place in the context of object-oriented frameworks. A description of this approach, along with an evaluation of its advantages and disadvantages is given in section 2.4.5. Before delving into the details of FAD, section 2.4.1 will provide an overview of the method.

2.4.1 Overview

FAD is part of the *quality-oriented software architecture design* (QASAR) method (fig. 2.14). This is an iterative method consisting of three phases, i.e. functionality-based architectural design (FAD), architecture evaluation and architecture transformation. QASAR provides support for an objective, rational design process, balancing and optimizing especially the quality requirements. The method iteratively assesses the degree up to which the architecture supports each quality requirement and improves the architecture using transformations, until all quality requirements are fulfilled.

The QASAR method performs architectural design focusing on the explicit evaluation of an design for quality requirements. Before the optimization of the architecture regarding quality requirements, a first version of the architecture based on functional requirements is constructed using the FAD method.

The FAD method consists of the following phases:

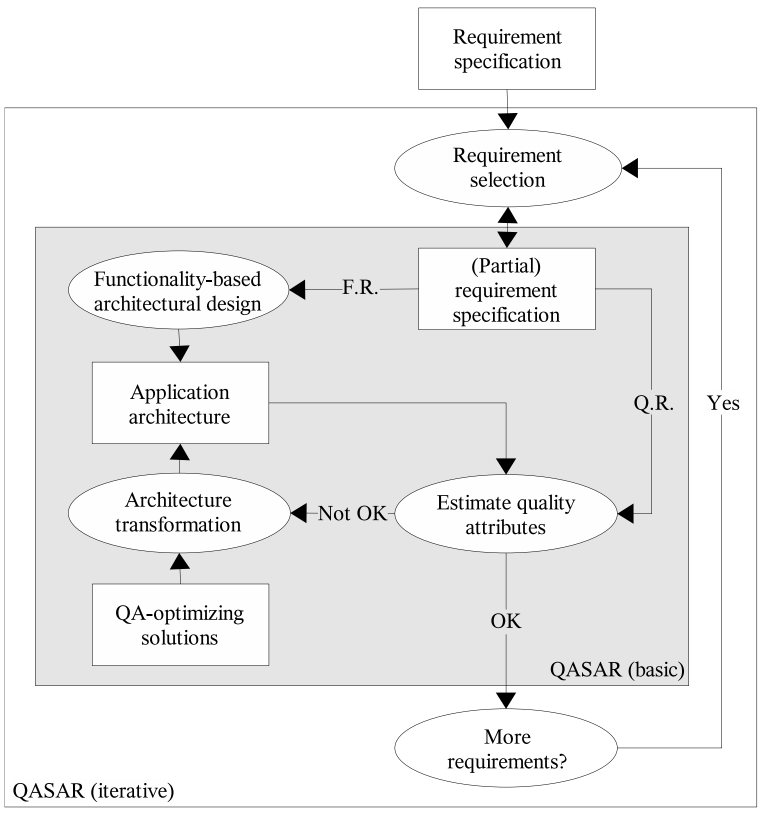

Requirement specification
Requirement selection
(Partial) requirement specification
F.R.
Functionality-based architectural design
Application architecture
Q.R.
Yes
Architecture transformation
Not OK
Estimate quality attributes
QA-optimizing solutions
OK
QASAR (basic)
More requirements?
QASAR (iterative)

Figure 2.14: The Quality-oriented Software Architecture design (QASAR) method

- Defining the system context
- Identifying the archetypes
- Decomposing the architecture into components
- Describing system instantiations

The artifacts produced during the FAD phases are shown schematically in fig. 2.15. Section 2.4.3 will provide a more detailed description of the first three FAD phases. The last FAD phase is merely a test to verify the scaleability of the developed architecture and does not directly influence the mapping between features and the architecture. Before proceeding with the description of the FAD phases, section 2.4.2 will present the case study used for the method's description.

2.4.2 Fire-Alarm PL Case Study

The FAD method has been applied in a variety of domains, e.g. the safety and security domain, the fire-alarm domain, the hemodialysis domain and the domain

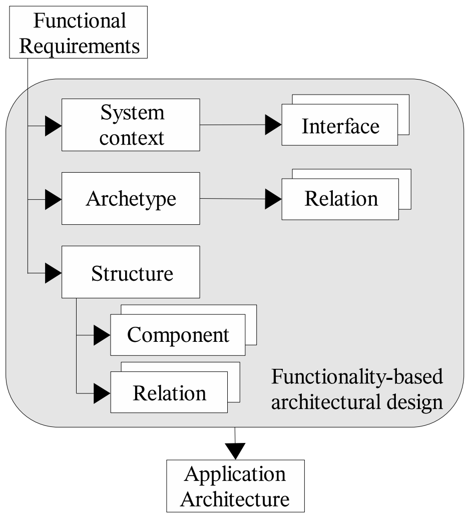


Figure 2.15: Artifacts of the FAD method

of operating systems for wireless devices. For the presentation and evaluation of the method, the fire-alarm PL case study has been selected from the book on FAD [Bos00]. This choice has been made to guaranty an unbiased evaluation context for the FAD method.

The main function of the fire-alarm system is to monitor a large number of detectors and upon the detection of a potential fire, to activate a number of outputs. Fire-alarm systems range from a few conventional smoke sensors with a three-led output, to high-end systems with sophisticated high-speed extinguisher control and a complex GUI. Finally, fire-alarm systems are highly distributed, i.e. detectors and outputs are distributed throughout a building or in the case of high-end systems, over several buildings.

2.4.3 FAD Phases

As shown in fig. 2.15, FAD produces an application architecture progressively, through a number of phases. During these phases the PL architectural components are developed. In order to be able to estimate the FAD's feature-architecture mapping, these phases are described in detail in the following sections. Eventually, section 2.4.4 will provide an evaluation of FAD's component development process with respect to feature-architecture mapping.

System Context

The first step of FAD performs an analysis of the context in which the system is to operate. From this perspective the externally visible behavior of the system is modeled. This is done through the definition of interfaces to external systems with which the PL software system has to interact.

The interfaces of the fire-alarm system to its context are shown in fig. 2.16. Interfaces are created between the software system and the physical detectors and outputs. Furthermore, an interface is created between the fire-alarm system and its operator. This enables e.g. the activation or deactivation of various parts of the system, monitoring of its behavior, etc. Finally, an interface between the building automation system and the fire-alarm system is added. This interface refers to operations needed to be performed, e.g. in the case of an alarm when a passage-control system is ordered to unlock all doors in a specific area.

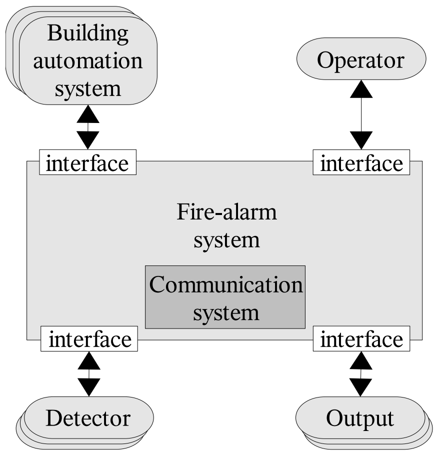


Figure 2.16: Interfaces of the fire-alarm system

Archetypes

After the definition of the system's context and the creation of the respective interfaces between the system and external entities, FAD performs the identification of *archetypes* and adds the relations between them.

Archetypes are the core abstractions of the system. They represent reoccurring, stable units of abstract functionality. The major parts of the system can be described with a small number of archetypes. These entities form the most stable part of the system and are seldom changed. Archetypes are instantiated in a large variety of ways to populate the system. By using the same fundamental concepts as a basis, archetypes promote the system's conceptual integrity [Bro95].

Identification of Archetypes

The identification of archetypes largely depends on the creativity and experience of the system architects. Therefore, FAD provides no concrete recipe for the identification of archetypes, but it does give some guidance. A starting point is a good understanding of the domain. Archetype identification should also not build up from concrete instances of the domain, like in the traditional object-oriented design methods, rather it should proceed in a top-down manner. That is, a holistic view of the whole system should be initially created upon which the archetypes are to be extracted. As the understanding of the domain increases, the architects often identify common characteristics between entities in disparate parts of the system. These entities may be a small structure or consist of a few entities. The reoccurring patterns are placed in a set of candidate archetypes.

Afterwards, an analysis of the potential archetypes is performed to filter out the system archetypes. Synonymous or largely overlapping structures are merged. Candidates with fundamentally different perspectives on the system should cause the removal of one of the two alternatives. Archetypes may also exist in different levels of abstraction, but the actual goal of the identification process is to achieve conceptual integrity.

Another important characteristic of archetypes is their relation to domain objects, i.e. features. Although archetypes can be modeled as domain objects, they are not found directly in the application domain. Instead, they are the result of a creative design process, that after analyzing the various domain entities, abstracts the most relevant properties and models them as architectural entities.

Relations between Archetypes

After the reduction of the set of potential archetypes, the relations between the archetypes are identified and defined. Relations between archetypes are generally domain specific and describe control and/or data flow. Relations often found in object-oriented modeling like generalization or aggregation should be avoided and lead to the reconsidering of the identified archetypes. For instance, a generalization relation should probably lead to the merging of the related archetypes. Nonetheless, such relations are not altogether excluded if necessary for the representation of semantically rich entities.

Fire-Alarm Archetypes

The archetypes for the fire-alarm system and their relations are shown in fig. 2.17.

- `Point` This is the highest abstraction modeling the fire-alarm domain functionality and presents the base abstraction for two subsequent archetypes.
- `Detector` This archetypes captures the functionality of the fire-detection devices, e.g. smoke and temperature sensors.
- `Output` This archetype represents the generic functionality for the various outputs of the fire-alarm system, e.g. extinguishing mechanisms, operator interfaces and fire notifications, e.g. to a fire-station.
- `Control Unit` Due to the distributed nature of the fire-alarm system, numerous groups of points control certain areas. Various groups need also to communicate with other groups in order to activate them, e.g. in the case of an alarm. The reoccurring abstraction capturing this crucial functionality, is a control unit interacting with the detectors and outputs of a group and also with other control units and their points.

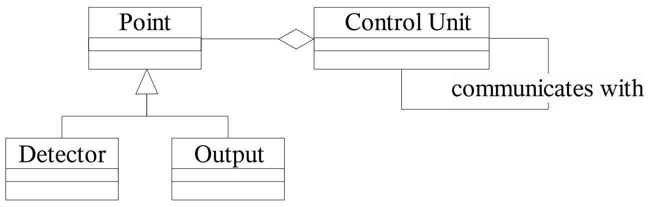


Figure 2.17: Archetypes of the fire-alarm PL and their relations

Component Decomposition

After the identification of archetypes, FAD decomposes the architecture into the actual architectural components and adds the relations between them. FAD identifies five sources of architectural component, along with two dimensions of system decomposition.

Component Identification & Specification

The sources for the architectural components are listed below:

1. *Interfaces* Based on this source, architectural components are identified based on the external interfaces of the system, e.g. for the fire-alarm system components should be identified to receive the interfaces for the interaction with the detector and output devices.

2. *Domains* Another source of components is the association of domains covered by the system with architectural components. Two types of domains are typically used, namely, application and computer science domains. The former relates to the problem domain covered by a system, e.g. for the fire-alarm system, components would be identified to model the fire-extinguisher domain, the sensor domain, etc. The latter domains refer to solutions needed to solve a problem from the perspective of the computer scientist, e.g. the fire-alarm system would be populated with components for the network communication protocols, user-interfaces, etc..

3. *Abstraction Layers* A third approach used in FAD is the decomposition of the system along horizontal layers implementing relevant functionality on different abstraction levels, e.g. components should be identified for the fire-alarm system within an application layer containing the alarm-signaling logic and communication logic and a hardware abstraction layer containing components for the lower level control of the output devices and network device drivers.

4. *Domain Entities* Yet another source of architectural components can be found in the domain knowledge of experts, reference literature, similar systems or existing standards in a domain. In this case, domain entities from the problem domain serve as a source for components, e.g. a distinguisher component or a smoke detector component would be created for the fire-alarm system.

5. *Archetype Instantiations* Since archetypes present recurring patterns in the system, they can also serve as input for the selection of components. For this purpose, they must be instantiated to concrete architectural components. The components of the fire-alarm system based on archetype instantiations are presented later in this section.

The dimensions along which the system decomposition in FAD takes place are functional versus entity-based decomposition and problem-domain versus solution-domain-based decomposition. Functional decomposition is concerned with the functions the system is to perform. A typical analogy are programs build with the C or Pascal programming languages. Entity-based decomposition refers to systems composed of the primary concepts defining a system. For instance, programs based on the C++ or Java programming languages use this approach or systems based on the component models of COM, CORBA and JavaBeans.

Along the second dimension of problem-domain versus solution-domain-based decomposition, systems can be decomposed according to the main functions or the entities of the domain. Architectural components from the problem domain are identified e.g. with the help of domain experts. Architectural components from the solution domain are identified, e.g. based on the experience of the system architects proposing specific software solutions.

Fig. 2.18 shows both decomposition dimensions with examples in each quadrant. For instance, a control theory architecture has architectural components representing the main concepts of the problem domain, e.g. feed-back and feed-forward control structures and entities based on mathematical concepts, like P, PI and PD controllers. Another example is the three-tire architecture for information systems on the lower-left quadrant. This architecture consists of components within layer from the solution domain, containing the main functions of the system, e.g. a graphical user interface layer, a application logic layer and a data storage layer.

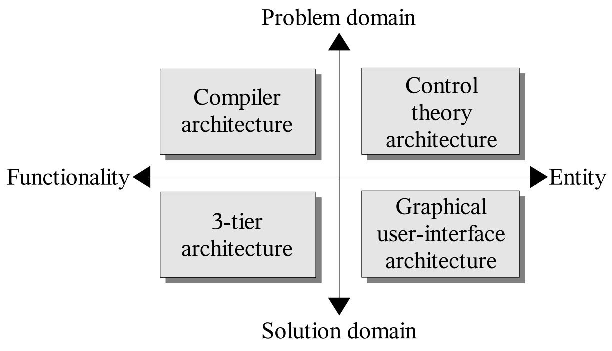


Figure 2.18: FAD dimensions of decomposition with examples

Component-Relation Identification & Specification

After the identification and specification of the architectural components, FAD progresses with the identification and specification of the component relations. Since the relations of the components are completely dependent on the specific software system at hand and its components, FAD provides a few guidelines and hints for the identification and specification of these relations.

According to FAD, relations can be identified on the basis of the component origin. The method gives two examples for the case of components defined in abstraction layers and components representing domain entities. Relations between components within abstraction layers can be identified depending on the abstraction levels to which they belong. Relations for components representing domain entities can be found in the domain models.

Furthermore, FAD suggests the scripting of *usage scenarios* for the identification of relations. During this process, the logical sequence of execution in the architecture is examined to identify possible communication between components.

Finally, FAD notes that relations at this stage of design should be kept at a minimum and should not be mapped to solution domain concepts, e.g. message send or pipe mechanisms, etc.. This clatters the design and is an indicator for low cohesion and high coupling in the software architecture.

Fire-Alarm Components

A few of the architectural components identified for the fire-alarm system are shown in fig. 2.19. The `Physical Point` components are instantiations of the `Point` archetype. The `Communication` component is a typical solution-domain entity. The `Section` components stem from the domain entity of the fire-alarm system domain. A `Section` component represents a controller along with the physical points monitored by it. The `Section` component is responsible for the detection and acting upon an alarm situation in a specific geographic area. The decomposition selected for the fire-alarm system is entity-based, with entities taken from both the problem and solution domains.

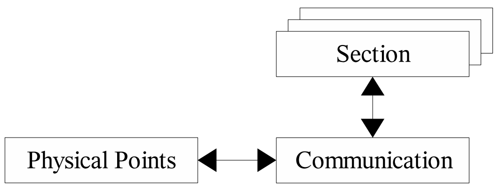


Figure 2.19: A partial view of the architectural components of the fire-alarm system

2.4.4 Evaluation of Component Development

As already mentioned, in order to achieve short times to market, the changeability and stability of the PL architecture is of primary importance. Since changes in PLs occur on the feature level, changeability and stability can be achieved if a strong mapping between the features and the architectural components exists. In the following sections, the FAD component development is examined from the perspective of feature-architecture mapping.

Sources of FAD Components

Section 2.4.3 presented the various sources of FAD components. These are interfaces, domains, abstraction layers, domain entities and archetype instantiations. The components identified from these sources can also be ordered across two dimensions: functional versus entity and problem domain versus solution domain.

Based on the definition of features presented in section 2.1, features "*represent an aspect valuable to the costumer*". This part of the definition denotes the fact that features are defined from the perspective of the PL costumers. Consequently, in order to achieve a stronger mapping between features and the architecture, the identified architectural components should also relate to the costumer's perspective.

The above considerations suggest that some of FAD's sources of components convey feature-architecture mapping, while others do not. More precisely, components based on the external interfaces of a system do support feature-architecture mapping. The interfaces that a system provides or requires for the communication with external systems originate from the externally visible properties of the system. The latter are also tightly related to the customer's perspective.

The exploitation of domains for the identification of architectural components can be further divided into problem and solution domains. Problem domains assist the development of components based on application concepts. These are conceived primarily from system experts, available literature or standards applied in the domain. Since system experts play an active role in the requirements specification of a PL, they also express the features of the system from the customer's point of view. Thus, component identification based on the problem domain enhances feature-architecture mapping. Components originating from the solution domain stem from the computer scientists building the PL. These components relate to concrete solutions understood by a software engineer, but they have little or no relation to the system features. Therefore, components based on the solution domain do not directly promote feature-architecture mapping.

The use of abstraction layers for the identification of architectural components is neutral with respect to feature-architecture mapping. That is because layers do not impose the nature of the architectural components within them. It may well be the case that architectural components within the various layers of the system still have a strong relation to features. Therefore, the critical issue in the use of abstraction layers is the actual nature of the system components and their distribution to the system layers.

Domain entities are ideal for a stronger mapping between features and the architecture. Since domain entities stem from the problem domain, the argumentation laid out above applies also here.

The last source of FAD's architectural components is the instantiation of archetypes. During the description of archetypes in section 2.4.3 it becomes clear that archetypes are not found directly in the domain. That is, archetypes have no direct relation to features, rather, they are abstractions of the common, recurring patterns throughout all domain entities. Nonetheless, archetypes instantiations may also produce components relating to the problem or the solution domain. In the former case, the derived components have a strong relation to features. In the latter case, the derived components increase the likelihood of feature scattering and tangling.

Fire-Alarm PL Example

For the fire-alarm PL case study, FAD identified the components shown in fig. 2.19. Fig. 2.20 shows a partial view of a possible structure for the fire-alarm PL FM, along with the mapping between the PL features and the FAD architectural components.

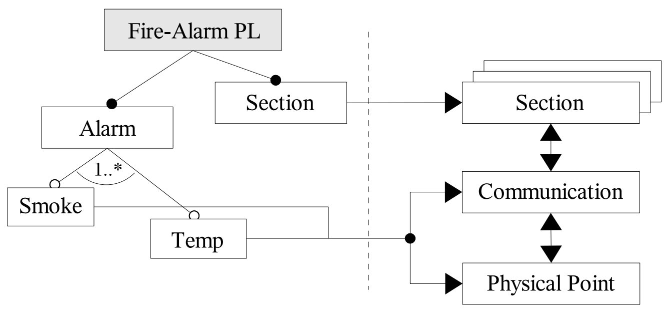


Figure 2.20: A partial view of the fire-alarm FM with the mapping between the PL features (left) and FAD architectural components (right)

The `Section` components directly implement the **Section** feature as shown by the arrow between the FM and the architecture. In this case there is no feature scattering or tangling, since the FAD components are derived from a problem domain entity. The `Physical Point` components are direct instantiations of the `Point` archetype. The `Communication` component stems from the solution domain. It enables the communication between the `Physical Point` and `Communication` components.

As shown in fig. 2.20, the **Smoke** alarm and **Temp** alarm features are scattered and tangled within these two components. More precisely, the `Communication` component is responsible for both the exchange of smoke and temperature data. For instance, it has to have knowledge of and be able to handle both particle-density data, as well as degrees Celsius data. This indicates the tangling of the aforementioned features. The `Physical Point` components respectively can detect both smoke and temperature alarms. They have for example the proper device drivers and algorithms for both smoke and temperature alarm detection. This indicates the tangling of the **Smoke** alarm and **Temp** alarm features within the `Physical Point` component.

The phenomenon of feature scattering becomes also evident, since parts of the features are implemented within two different components. For example, a smoke alarm is first identified in a `Physical Point` component and is then propagated through the `Communication` component to the proper `Section` component. The same holds for a temperature alarm.

The scattering and tangling of the **Smoke** alarm and **Temp** alarm features deteriorates the changeability and stability of the PL software architecture. For instance, a change on the data transmission protocol of particle-density data from a hardware sensor would trigger unpredicted changes on the **Communication** component. These in turn could trigger changes in the implementation of data transmission for a temperature alarm. Since one component is used for both kinds of alarm, one must depend alone on the good design and programming of the internals of the `Communication` component. Nonetheless, a clear separation of the implementation of both the smoke and temperature alarm is not guarantied.

A possible architecture that would enable a stronger mapping between the features and the architecture is shown in fig. 2.21. The **Smoke** feature is implemented in each of the `Smoke Alarm` components. The latter are responsible for controlling the sensor hardware and implementing the algorithm for signaling an alarm. A `Smoke Alarm` component is also responsible for transmitting particle-density and alarm data to the right section. The `Temp Alarm` components respectively implement the **Temp** feature.

The architecture shown in fig. 2.21 causes no feature scattering or tangling. The alarm detection functionality, the communication functionality and the knowledge of the proper section to notify, all belong respectively to the specification of the **Smoke** and **Temp** features. Exactly this functionality is placed within separate

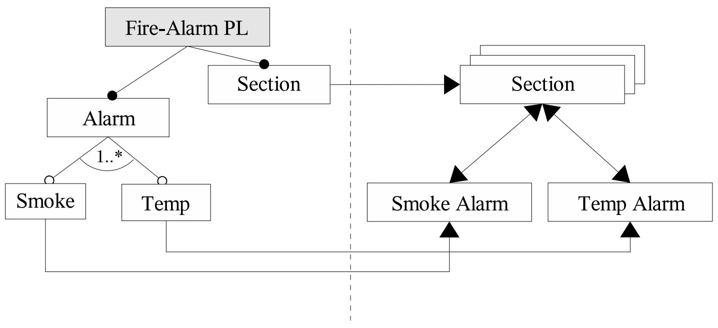


Figure 2.21: A more direct mapping between the fire-alarm features (left) and the architecture (right)

components for each of the features. The FAD architecture (fig. 2.20) on the other hand, scatters and tangles the smoke and temperature alarm detection functionality to each of the `Physical Point` components and the communication functionality and knowledge of the proper section to the `Communication` component.

A change in one of the alarm features for the FAD architecture (fig. 2.20) would cause unpredictable, cascading changes to the FAD architectural components. The architecture allowing a stronger mapping is more resistent to feature-level changes. For instance, the particle-density communication protocol is now implemented within the `Smoke Alarm` component. Since the communication protocol for the **Smoke** feature is separated from the communication protocol for the **Temp** feature, the change would not propagate to the other components.

2.4.5 Object-Oriented Frameworks

As already mentioned, minimal effort should be required for the development of the PL products, i.e. products should be directly generated from the PL assets. FAD suggests the use of object-oriented frameworks for product instantiation. The same practice is also followed by the KobrA [A$^+$02] PL method. The following sections will provide a detailed description and an evaluation of this approach regarding the generative product instantiation.

Framework Concepts

The most often referenced definition for object-oriented frameworks is found in [JF98]:

"*A framework is a set of classes that embodies an abstract design for solutions to a family of related problems.*".

Based on the above definition, a framework is a partial design and implementation for an application in a given domain. Applications are constructed by using the framework as a basis and extending it with application-specific functionality. Moreover, a framework consists of a framework architecture, specifying a number of abstract classes and possibly concrete classes inheriting from these abstract classes that provide reusable implementations.

The notion of object-oriented frameworks was originally tightly related to a single framework used for the construction of applications. Nowadays, multiple object-oriented frameworks are used in the construction of software systems. Each framework covers a domain and since systems tend to cover multiple domains, multiple frameworks are needed to cover the required system functionality. The ability of object-oriented frameworks to cover the functionality of a domain also makes them suitable for use with software PLs.

In order to better argue on object-oriented frameworks, a number of concepts must be introduced. Namely, the concepts of *core framework design*, *framework internal increment* and *application-specific increment.*

Core Framework Design The core framework design comprises both abstract and concrete classes in the domain. The concrete classes are transparent to the framework user, e.g. basic storage classes. Abstract classes are either transparent or are intended to be sub-classed by the framework user. The core framework design describes the typical software architecture for applications in the domain.

Framework Internal Increment These are additional classes in the form of libraries. They capture the common implementation of the core framework design and intend to make it more usable. The common categories of internal increments are:

- **sub-classes** representing common realizations of the concepts captured by the super-classes, e.g. for an abstract `Device` class there may exist a number of concrete sub-classes for each commonly used real-world device.
- **a collection of sub-classes** representing the specifications for a complete instantiation of the framework in a particular context.

Application-specific Increment As mentioned above, an application is composed of multiple frameworks, i.e. an application is composed of one or more core framework designs, each framework's internal increments and application-specific classes and objects. The latter comprise the application-specific increment.

Object-oriented frameworks can be categorized into *white-box* and *black-box* frameworks or *calling* and *called* frameworks. White-box (inheritance-based) frameworks are customized through sub-classing of the framework classes. Black-box (parameterized) frameworks are customized by using different combinations of the framework classes. Black-box frameworks require a deeper understanding of the variable aspects of the domain compared to white-box frameworks. Due to this predefined flexibility, black-box frameworks are more rigid in the domain they implement. A calling framework is an active entity invoking parts of the applications created by it. Called frameworks on the other hand, are passive entities that can be invoked by its applications.

In practice, frameworks cannot be definitely categorized as pure white-box or black-box frameworks or as calling and called frameworks. Parts of a framework are extendable through sub-classing, others can be parameterized, while some parts of the framework call an application and other parts are called by an application.

FAD utilized object-oriented frameworks for the instantiation of PL products. It identifies four different framework component models for this purpose. The criteria used for the distinction between these models are:

1. the amount of application-specific functionality required by a framework

2. the organization of the framework with respect to the number of independent variation points

Each of the models is examined in the following sections.

Product-specific extension model

This framework component model covers only the common behavior for all products in the PL. Each product is instantiated by extending the framework with product-specific extensions. Fig. 2.22 shows a graphical illustration of such a framework. The core framework design comprises of a set of operations, $\mathtt{o_1}, \mathtt{o_2}, ..., \mathtt{o_n}$ and a set of interface types, i.e. $\mathtt{i_1}, \mathtt{i_2}, ..., \mathtt{i_n}$. The operations may return references to objects of the specified interface types that are then used for the continued operation. Ideally, the framework interface is not affected by product-specific extensions. However, extending or changing the framework interface cannot be avoided for specific problems.

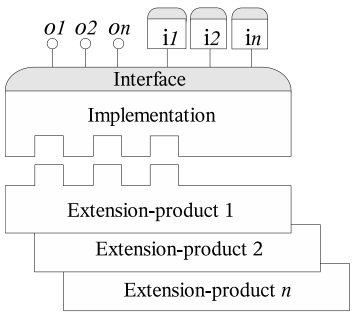


Figure 2.22: Product-specific extension model

Standard-specific extension model

In this model, each standard, e.g. a file-system or communication protocol is implemented as an extension to the framework. Each product incorporates one or more framework implementations, either as product variants or as configurable parts of the product. The common part of the framework only defines the framework interface and no or very little common behavior between the framework implementations. Fig. 2.23 shows a schematic illustration of a standard-specific extension framework.

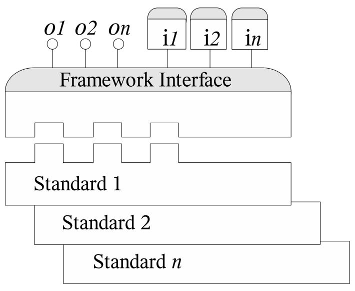


Figure 2.23: Standard-specific extension model

Fine-grained extension model

The two previous component models extend the object-oriented framework with a single extension that covers all variation points of the framework. The fine-grained extension model takes the opposite approach. It provides small modules that only cover one or a few variation points and that are themselves configurable. In this case, the common framework consists of an interface and an implementation common to all instantiations. For each variation there exists a set of generic extensions, which can be configured with product-specific extensions. Fig. 2.24 presents a fine-grained framework graphically.

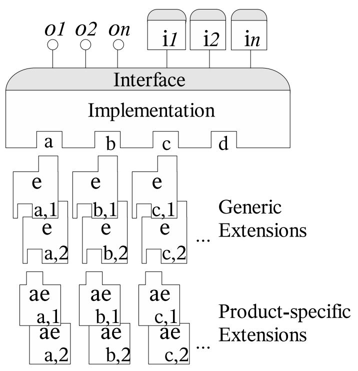


Figure 2.24: Fine-grained extension model

Generator-based model

The generator-based model is practically an extension of the fine-grained extension model. Once the suitable variation points and the useful extension of a framework have been identified, this model makes use of a generator for product instantiation. The latter can be either a graphical configuration tool in which components are configured with available extension components or a domain specific language (DSL) in which a configuration is specified and afterwards a matching component is generated. Fig. 2.25 shows the generation process for this model.

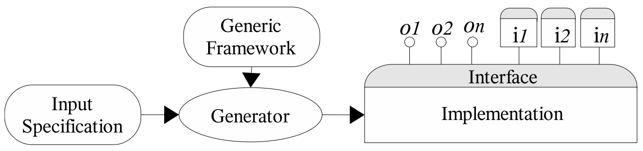


Figure 2.25: Generator-based model

2.4.6 Evaluation of Framework Component Models

This section provides a general evaluation of the various framework component models supported in FAD, from the perspective of the effort needed for product instantiation.

Product-specific extension model The primary advantage of this approach is its simplicity. Since each product is an independent extension of the framework, a relatively simple organization is needed for the development and maintenance of the products. The main disadvantages of the component model is its lack of reuse between product-specific instantiations and its inflexibility. More precisely, there may exist common requirements for a number of product instantiations. Because each product is an independent module, the common parts of the products cannot be reused. Finally, this framework model is rather inflexible, since a change occurring in the framework has a direct impact on all products.

Standard-specific extension model One advantage of this model is the uniformity of the provided interface. This allows all framework components to be readily able to communicate with every other component of the framework. Another advantage of this model is the evolvability of the various framework implementations. Namely, as long as each framework implementation adheres to the same framework interface it can evolve independently from the others. One disadvantage of the model is lack of reuse of the commonalities between the various framework implementations. Furthermore, the model does not allow for product-specific extensions and it illustrates decreased maintainability, since changes to the component interface are propagated to each other implementation of the framework.

Fine-grained extension model One advantage of this model is its configurability. The framework user is free to compose arbitrary sets of extension components for its specific product. Another advantage is the reusability of the framework extensions. This is due to the relative atomicity of the extensions. Unfortunately, this model illustrates high complexity, depending on the number of variation points, extension components and the relations between them. Another issue is the difficulty of achieving the proper granularity for the extension components and thus avoiding e.g. the creation of too fine-grained components.

Generator-based model This model has the same advantages as the fine-grained extension model. Namely, flexibility and configurability. Additionally, this model also manages the complex use of the fine-grained extensions of the framework. The latter automates the product instantiation process and reduces their time-to-market. The disadvantages of this model are listed below:

- In order to define the right fine-grained extension components and identify the required variation points, the generator-based model requires a mature domain. In many cases developers are confronted with immature domains for which the generator-based model is unsuitable.

- The generator-based model illustrates decreased evolvability. More precisely, the framework is more expensive to evolve since every change to the framework requires respective changes to the configuration tool or DSL.
- The use of the generator-based model limits the number of possible products to be combined. This is due to the fact that the tool or DSL can only generate the combinations imagined and intended by its designer. Extending the tool or DSL is again related to additional cost and effort.
- Finally, because of feature scattering and tangling, the generator-based model may not enable the mapping of a feature to the extension components or the mapping may be inefficient and introduce a lot of unwanted functionality. This may in turn influence the product's performance, price, etc..

To summarize, the product-specific and standard-specific models are simple to use in contrast to the fine-grained extension model, but suffer from decreased maintainability and lack of reuse. The fine-grained extension and generator-based models are much more flexible than the first two models. The generator-based model also partially solves the complexity of the fine-grained extension model.

The primary advantage though of the generator-based model is enabling a generative approach to application engineering, something that is not supported by the other models. This is a vital precondition for the minimization of product instantiation effort. Providing solutions to the disadvantages of the generator-based model is the goal of this work.

2.4.7 Conclusions

This section has evaluated the FAD PL method from the perspective of the identified open issues of feature-architecture mapping and product instantiation.

Concluding the evaluation of the FAD component development, one can say that the use of the system's external interfaces and problem domain entities as source for components does promote feature-architecture mapping. This is also true for archetype instantiations that produce components closely related to problem domain entities. The use of abstraction layers has been found to be neutral with respect to feature-architecture mapping. Solution domain entities and direct instantiations of archetypes have a rather negative influence to the mapping between features and the architecture.

Nonetheless, the use of solution domain entities may be unavoidable in certain occasions. For instance, there may exist hard architectural requirements that are not visible to the customer and therefore not present in the FM. In such cases, components must be developed that have little or no relations to customer-visible features. Additionally, implementation issues may require components from the solution domain.

As mentioned above, the direct instantiation of archetypes does not directly support feature-architecture mapping. Nevertheless, the consistent use of archetypes throughout the software system assures another valuable system property, namely, conceptual integrity [Bro95]. The latter promotes the system's understandability and maintainability.

With respect to product instantiation, FAD's generator-based model stands out among the four different framework component models. On the one hand, the generator-based model allows for a generative approach to application engineering in contrast to the other three models. On the other hand, it suffers from a number of problems (sect. 2.4.6), which need to be solved in order to achieve an efficient product generation. Solving the open issues of the generator-based model is one of the goals of this work.

2.5 Generative Programming Techniques

Generative programming techniques can also be utilized as alternative solutions to feature-architecture mapping and feature-level variability, which are both identified as open issues in the general state of the art analysis (sect. 2.2). At first, a brief presentation of the most established generative programming techniques, along with their integration in PL methods will be given. Based on this, a representative method will be selected for more detailed examination.

Aspect-Oriented Programming Aspect-oriented programming (AOP) is a broadly accepted generative programming technique. AOP aspects are concerns, e.g. concurrency, distribution or persistence and are expressed with a special purpose programming language. During compilation, the aspect code is weaved into the source code of the existing system, thus extending it to support new concerns. AOP offers an implementation for Java, namely, the AspectJ tool [Kic01] and has been extended to support the UML [SY99]. Up to this point there exists no explicit integration of AOP with a PL method.

Hyperspace The Hyperspace approach [OT01] maps any kind of artifact, e.g. requirements specifications, architectural entities or source code, to any concern that a system stakeholder may have, e.g. to functional or quality features. This is implemented through a specifically developed language that maps e.g. a feature to the various classes and their members in a software system. The Hyperspace approach has an implementation for the Java programming language, namely, the HyperJ tool [TO01] and has been extended to map to UML design artifacts, e.g. use case diagrams, collaboration diagrams, etc. It has been successfully integrated in the FeatuRSEB method, yielding Hyper-FeatuRSEB [Boe02]. Another well established PL development methodology based on similar principles is the GenVoca method [BG97].

As shown from this brief description of the most important generative programming techniques, the Hyperspace approach has a wider spectrum of application compared to AOP. Furthermore, it has been explicitly integrated into the FeatuRSEB PL method. The basic principles of the Hyperspace approach are shared by the other generative programming techniques. Because of these reasons, this work will examine the Hyperspace approach in more detail and will evaluate it from the perspective of feature-architecture mapping and feature-level variability.

2.5.1 The Hyperspace Approach

The Hyperspace approach has been developed in the IBM research center. It is geared towards multi-dimensional separation of concerns. The concept of separation of concerns [Par72] has been first introduced by Parnas in his work on the decomposition of software systems. A software keeping concerns separate from each other can achieve high changeability and reusability. In the Hyperspace approach, the "tyranny of the dominant dimension" is introduced. This relates to the fact that software systems are built to comply to one single dimension. For example, functions in procedural-oriented programming, like Pascal and C or classes in object-oriented programming, like Java and C++. The Hyperspace approach strives the encapsulation and thus the separation of all possible concerns in a software system. For instance, a system in the Hyperspace approach can be decomposed based on its requirements specification, its features or any other concern a stakeholder of the system may have.

A Simple Software Engineering Environment

The concepts of the Hyperspace approach can be explained more easily in the context of an example. The example used is a simple Software Engineering Environment (SEE) adapted from a publication on the Hyperspace method [OT01]. The source code and the Hyperspace artifacts can be downloaded from the Hyperspace homepage [Tar05]. The SEE requirements specification are as follows:

- The SEE shall allow the definition of a mathematical expression
- The SEE shall allow the evaluation of a mathematical expression
- The SEE shall allow the textual display of a mathematical expression

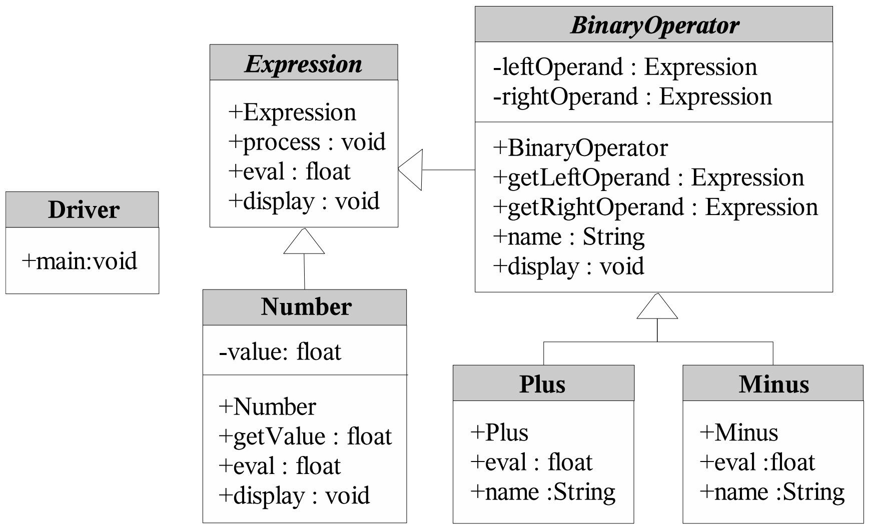


Figure 2.26: A simple Software Engineering Environment architecture

The architecture of the SEE (fig. 2.26) is based on an Abstract Syntax Tree (AST) representation and defines a class for each kind of AST node. The function of the SEE can be understood by following the creation, evaluation and display of a simple expression. Consider the expression `(1 + 1)`:

The `Driver` object is instantiated by the Java virtual machine and the `main()` method is called, as shown in listing 2.1. The expression created consists of the concrete `Plus` binary operator object and two instances of the `Number` class instantiated with the value `1`. The constructors of the classes involved and the `process()` method are shown in listing 2.2.

At first, because of the class hierarchy, the constructor of the `Expression` class is called twice, followed by subsequent calls to the constructor of the `Number` class with the value `1`. In the `Number` class constructor the `_value` instance variable is set to `1`. Then again the `Expression` constructor is called, followed by a call to the constructor of the `BinaryOperator` class, which sets the inherited instance variables, `_leftOperand` and `_rightOperand` to the new `Number` objects. Note, that the `BinaryOperator` constructor takes `Expression` objects as parameters. It is possible to use instances of the `Number` class with the constructor because `Number` is also a subclass of `Expression`. Immediately after the call to the `BinaryOperator` constructor, the constructor of the `Plus` object is called which takes no further actions.

Listing 2.1: The `main()` method

```java
public static void main(String[] args) {
    Expression expr = new Plus(new Number(1),
                               new Number(1));

    expr.process();
}
```

After the expression has been created, the `process()` method is called. The latter is implemented in the `Expression` class and it evaluates and displays the expression. Note that the `System.out` is a Java output stream.

Listing 2.2: Methods involved in the creation of an expression

```java
public class Expression {
    public Expression() { }
    public void process() {
        eval();
        display(System.out);
    }
}
public Number(float value) {
    _value = value;
}
public BinaryOperator(Expression left,
                      Expression right) {
    _leftOperand = left;
    _rightOperand = right;
}
public Plus(Expression left,
                Expression right) {
18      super(left, right);
   }
```

The classes and methods taking part in the evaluation of the expression are shown in listing 2.3. Due to polymorphism, when the `eval()` method within the `Expression` class is called (listing 2.2, line 4), actually, the `eval()` method of the `Plus` object is called (listing 2.3, line 2). In this method, the `getLeftOperand()` and `getRightOperand()` methods of the `BinaryOperator` class are called. These return objects of the class `Expression`, which are actually the `Number` objects set in the private instance variables `_leftOperand` and `_rightOperand`, during the creation of the expression. Upon these objects the `eval()` method is called. Again because of polymorphism, the `eval()` methods of the `Number` objects are actually called. These return the value stored in the `_value` instance variables of the objects, which is `1`. The return values are added, yielding the right result, i.e. `2`, which is then returned as the result of the evaluation.

Listing 2.3: Methods involved in the evaluation of an expression

```
1  public class Plus {
       public float eval() {
3          return(getLeftOperand().eval() +
                   getRightOperand().eval());
5      }
   }
7  public class BinaryOperator {
       public Expression getLeftOperand() {
9          return _leftOperand;
       }
11     public Expression getRightOperand() {
           return _rightOperand;
13     }
   }
15 public class Number {
       public float eval() {
17         return(_value);
       }
19 }
```

After the expression has been evaluated, the `display(System.out)` method within the `process()` method of the `Expression` class is called (listing 2.2, line 5). The class and methods involved in displaying the expression are shown in listing 2.4. The polymorphic method `display(PrintStream displayDevice)`, defined in the `BinaryOperator` class, is initially called (line 2). It displays a left bracket "`(`" and then calls the polymorphic `display(...)` method of the `Number` class. This prints the context of the `_value` instance variable, which has been set to `1`. Following this, a call to the `name()` method of the `Plus` object is made. This returns the symbol "`+`" for the addition operator, which is printed within the `display(...)` method of the `BinaryOperator` class. Finally, the `display(...)` method of the right operand is called, i.e. the `Number` object and the left bracket is displayed. The result of the whole display process is as expected: `(1 + 1)`.

Listing 2.4: Methods involved in displaying an expression

```
1  public class BinaryOperator {
       public void display (PrintStream displayDevice) {
3          displayDevice.print  ( "(" );
           getLeftOperand().display (displayDevice);
5          displayDevice.print  ( "␣" + name() + "␣");
           getRightOperand().display(displayDevice);
7          displayDevice.print  ( ")" );
       }
9  }
   public class Number {
11     public public void display (PrintStream displayDevice) {
           displayDevice.print(_value);
13     }
   }
15 public class Plus {
       public String name() {
17         return "+";
       }
19 }
```

Hyperspace Concepts

The SEE is a typical example to illustrate the effect of multiple concerns within a software system. The SEE architecture is decomposed based on classes, i.e.

`Expression`, `Number`, etc., while the requirements specification have defined the system based on features, i.e. creation, evaluation and display of a mathematical expression. The latter are implemented throughout the entire class hierarchy. This illustrates the effect of feature scattering and tangling.

The Hyperspace approach attempts to resolve feature scattering and tangling by encapsulating the various methods and instance variables in a so called *hyperslice*. A hyperslice should encapsulate exactly one concern. Figure 2.27 shows the decomposition of the SEE architecture based on its features, through hyperslices.

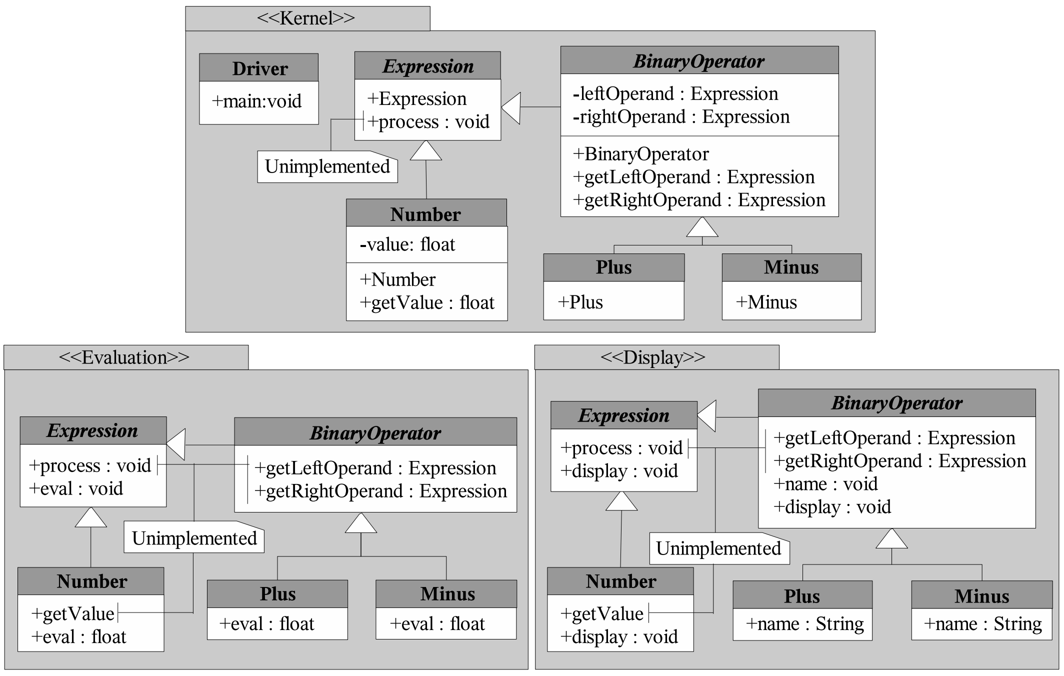


Figure 2.27: A decomposition of the SEE architecture based on its features through hyperslices

The evaluation feature of the SEE is encapsulated in the `Evaluation` hyperslice. This contains all parts of the system related to the evaluation of an expression. These would normally be all `eval()` methods in the class hierarchy. Nonetheless, the hyperslice contains also the `getLeftOperand()` and `getRightOperand()` methods of the `BinaryOperator` class, the `getValue()` method of the `Number` class, as well as the `process()` method of the `Expression` class. Additionally, as the UML notes imply, these methods are *unimplemented*. Only the definition, not the implementation of these methods is added to the hyperslice to assure its *declarative completeness*.

Declarative completeness is a special concept in the Hyperspace approach. It means that a hyperslice must at minimum include a declaration for every function that any of its members invokes or any variable its members use. The hyperslice need not provide an implementation for these declarations. The Hyperspace approach suggests that declarative completeness eliminates coupling between hyperslices. Instead of one hyperslice referring to another, thereby depending upon the other specific hyperslice, each hyperslice states what it needs by means of the abstract declarations, thereby remaining self-contained. The implementation of these abstract declarations can be provided by any appropriate hyperslice(s) through integration.

The `Display` hyperslice decomposes the system in a similar way as the `Evaluation` hyperslice. In the SEE example, the implementation for the accessor methods in the hyperslices is provided by the `Kernel` hyperslice. The latter contains all parts of the system that are responsible for the creation of an expression, as well as those parts of the system implementing its basic capabilities. The constructor, accessor and modifier methods, along with the instance variables, allow the creation of an expression. The `Driver` class with the `main()` method, along with the `process()` provide the system basic capabilities.

Note that the `process()` method in the `Kernel` hyperslice remains unimplemented. This special handling of the `process()` method is typical in the Hyperspace approach. As shown in listing 2.2, the `process()` method drives both the evaluation (line 4) and the displaying (line 5) of an expression. In order to achieve a clear separation of concerns, line 4 should be included in the `process()` method of the `Evaluation` hyperslice and line 5 in the `process` method of the `Display` hyperslice. However, the current implementation of the Hyperspace approach treats methods as primitive units, which means that it does not support the mapping of individual statements to concerns. Since the method can not be ”torn apart”, it is entirely excluded from all features.

As mentioned earlier, an implementation of the Hyperspace approach is provided for Java, namely, the HyperJ tool [TO01]. HyperJ defines a special language to decompose the architecture into concerns. Initially, a definition of the *Hyperspace* must be provided. Listing 2.5 shows the Hyperspace definition for the SEE. This simple definition specifies that all classes within the package `tu.ilmenau.SEE` should be included in the Hyperspace. When HyperJ runs, it will automatically create one dimension, i.e. the ClassFile dimension and one concern in that dimension for each class. The contents of those concerns are the *units* (interfaces, classes, methods, and member variables) in the corresponding classes.

Listing 2.5: The SEE Hyperspace

```
hyperspace HyperspaceSEE
```

```
    composable class tu.ilmenau.SEE.*;
```

In order to map the concerns, i.e. the features of the SEE system to its architecture, i.e. its classes, HyperJ uses a *concerns mapping* shown in listing 2.6. The concerns mapping starts with assigning the entire system to the `Feature.Kernel` hyperslice. The `package` statement indicates that all classes, along with their methods and instance variables are to be included in the `Kernel` feature of the `Feature` dimension. Exceptions to this main rule are provided in the following lines. For instance, in line 2, all `eval()` methods in the system are included in the `Evaluation` hyperslice. The HyperJ tool then automatically adds abstract declarations to all methods referenced, but not declared within the hyperslice, as shown in figure 2.27. As mentioned above, the `process()` method drives both the evaluation and displaying of an expression. Because it can not be taken apart, it is assigned to the `Feature.None` concern. The latter is a Hyperspace-specific concern used in situation where concerns are intermingled within a primary unit, e.g. a class method.

Listing 2.6: The SEE concerns mapping

```
1 package tu.ilmenau.SEE  : Feature.Kernel
2 operation eval          : Feature.Evaluation
3 operation display       : Feature.Display
4 operation name          : Feature.Display
5 operation process       : Feature.None
6
```

A final product can now be composed through a mix-and-match of features. Such a product is referred to as a *hypermodule*. A hypermodule is created through the composition of hyperslices by means of *composition rules*. There exist a large number of composition rules for hyperslices. A complete list can be found in the HyperJ manual [TO01]. Two representative examples of composition rules are illustrated below. Listing 2.7 shows a hypermodule definition that includes only the evaluation feature of the SEE system.

Listing 2.7: The SEE Evaluation hypermodule

```
hypermodule EvaluationSEE
    hyperslices:
        Feature.Kernel,
        Feature.Evaluation;
    relationships:
        mergeByName;
        equate operation Feature.Kernel.process,
                         Feature.Evaluation.eval;
end hypermodule;
```

In this hypermodule, the `Kernel` and `Evaluation` concerns are related by a `mergeByName` composition rule, referred to as an integration relationship. The "By-Name" indicates that units in the different concerns are considered to correspond if they have the same names and signatures, where appropriate. The "merge" indicates that corresponding entities are to be combined so as to include all their details. For example, the `getLeftOperand()` and `getRightOperand()` methods in the `Evaluation` hyperslice (fig. 2.27), which have no implementation, will be merged with the implemented methods of the `Kernel` hyperslice, thus providing the proper functionality in the final product.

The second integration relationship, `equate`, accomplishes the special handling of the `process()` method. As discussed earlier, the `process()` method was excluded from the hypermodule by delegating it to the `Feature.None` concern (listing 2.6, line 6). However, the `Driver` calls it (listing 2.1, line 5) within the `Feature.Kernel` concern (fig. 2.27). During declaration completion, to make `Feature.Kernel` a valid hyperslice, HyperJ inserts an abstract declaration of `process()`. In this hypermodule, it is specified that the abstract declaration must be bound to the `eval()` method of the evaluation feature. The `equate` relationship does just that. It assures that when the Driver calls `process()` at runtime in the composed hyperslice, only the `eval()` method is called.

This hypermodule definition yields an executable version of the SEE system that contains only the evaluation feature. Respectively, it is possible to define a hypermodule having only the display feature of the SEE.

2.5.2 Evaluation of the Hyperspace Approach

From the description of the Hyperspace approach, one can conclude that the method provides concrete mechanisms for the mapping between features and the architecture. Its integration with the FeatuRSEB method, yielding the HyperFeatuRSEB [Boe02], indicates also its applicability in the instantiation of PL products and thus its contribution to the variability on the feature level. These two aspects of feature-architecture mapping and feature-level variability will be evaluated in this section.

Feature-Architecture Mapping

The problems of the Hyperspace approach, from the perspective of feature-architecture mapping, lay mainly on the introduction of extra artifacts and the effort needed for their development and maintenance, as well as on the interaction between hyperslices.

As shown in the SEE example, various artifacts were required for the achievement of a decomposition based on features. At first, one needs to define a Hyperspace that contains all architectural entities in the system. Afterwards, the system must be decomposed through the definition of hyperslices. Finally, an executable version of the system must be created through the definition of hypermodules. The development of these artifacts for the SEE was rather simple. Nonetheless, as shown in the case of the `process()` method, intermingled features within a class method cannot easily be separated. More precisely, the `process()` method was not implemented within the hyperslices, rather it was assigned to the `Feature.None` concern and the `equate` integration relationship had to be used to allow a mapping to the desired feature at compile time.

What was actually required, was the placement of the calls `eval()` and `display(System.out)` of the `process` method (listing 2.2, lines 4 and 5) into the `process()` methods of the `Evaluation` and `Display` hyperslices (fig. 2.27) respectively. However, assigning source code lines to features, despite the fact that it is rather cumbersome, it would also dramatically deteriorate the maintenance of both the system and the hyperslices. The solution provided by the Hyperspace approach might have been sufficient for the SEE example, but it is doubtable if it would suffice for large industrial PLs.

Another open issue with respect to feature-architecture mapping is the hyperslice interaction. The authors of the Hyperspace approach suggest that by making hyperslices declarative complete, a decoupling of the hyperslices is also achieved (sect. 2.5.1). Nevertheless, hyperslices are in fact as decoupled as the architecture of the system itself. For instance, one cannot treat the `Evaluation` hyperslice independently from the other hyperslices. A change, e.g. in the implementation of the `getLeftOperand()` method, would propagate to all other hyperslices of the system, since their proper operation depends on that method. In order to guarantee the proper integration of the changed hyperslice with other hyperslices, a developer should take into consideration all points in the system where this method is needed. In an actually decoupled system, as long as the interface of the method does not change, i.e. both syntactically and semantically, changing the method's implementation should have no effect on the rest of the system. This is not true in a "hypersliced" system.

In the SEE example, a change of the internal implementation, e.g. of the `getLeftOperand()` method in the `Evaluation` hyperslice would cause the change of the two other hyperslices, i.e. the `Kernel` and `Display` hyperslices. In other words, a change in one feature would cause the change in two out of two features in the SEE system. An analysis on the source code of a PL developed with the HyperFeatuRSEB method [Boe02] has been performed in the context of this work [SRP04]. The analysis was based on the number of unimplemented methods found in the system's source code. This is possible because of a special exception thrown in methods implemented within other hyperslices:

```
throw new com.ibm.hyperj.UnimplementedError();
```

This exception is an indicator for hyperslice interaction. The results of the source code analysis showed that the PL had 1243 such unimplemented methods from overall 4197, shared between various combinable features. More precisely, 1 out of 3 method changes would cause at least 2, at most 19 and on average 4 features to change.

Feature-Level Variability

Regarding feature-level variability, the generative programming techniques fall in the category of the code fragment superimposition variability mechanisms. These mechanisms are primarily applicable for the selection of one variant from a number of alternatives at compile time.

As shown in the SEE examples, a feature can be selected from a number of alternatives through the integration of one or more hyperslices. The latter takes place upon product compilation. The various methods in the classes of hyperslices are merged together to form the final product. Apart from merge, there exist also other integration relationships, e.g. override.

From the perspective of feature-level variability, the Hyperspace approach incorporates extra artifacts for the isolation of the variants. The main problem of this approach has already been discussed in the context of feature-architecture mapping. Namely, high effort may be required for the isolation of the variants in a system where a high degree of feature tangling is present. In some cases, it may not be possible to isolate the variants at all.

In the example of the `process()` method of the SEE system, if the tangling of the features was more intensive, namely, within one code line, then it would have been impossible to separate one feature from the other. In large systems, the probability of such scenarios is quite high. Nevertheless, the maintainability of the Hyperspace artifacts is more advanced compared to the other variability mechanisms (sect. 2.3.3).

2.5.3 Conclusions

The generative programming techniques provide concrete approaches for the resolution of the issues of feature-architecture mapping and feature-level variability. They have been integrated into established PL methods to enhance variability on the feature level and thus, to reduce the time-to-market of the PL products. This work provided a detailed examination of a representative generative programming technique, namely, the Hyperspace approach.

With respect to feature-architecture mapping, the Hyperspace approach does not provide a unambiguous separation of concerns. The hyperslice interaction remains unresolved and is treated rather on a syntactical level. The extra artifacts introduced also add extra effort for their creation and maintenance.

Regarding feature-level variability, the Hyperspace approach shares more or less the same problems with variability mechanisms of the same level (sect. 2.3.3). It needs the introduction of extra artifacts, whose development effort is highly dependent on the degree of feature scattering and tangling. As shown in the SEE examples, for small-sized variants illustrating low tangling, e.g. within a class, the Hyperspace approach can be quite efficient. For large features, where the probability of tangling increases, isolating a specific variant is at least proportional to the effort needed to resolve feature tangling.

Nonetheless, if a certain degree of decoupling is present in the architecture, the Hyperspace approach can be an efficient way to provide a mix-and-match of features. Additionally, in the case of monolithic systems, were the resources for redesign are not available, the Hyperspace approach can be applied to ease the evolution of the system. Finally, one of the strengths of the Hyperspace approach is its enormous flexibility to provide a mapping between the architecture and any conceivable concern of any system stakeholder simultaneously.

2.6 Motivation and Objectives

The primer motivation for this work is the enhancement of feature-architecture mapping in the context of PLs. The FeatuRSEB, FAD and the Hyperspace approach were selected as mature and representative state of the art solutions to feature-architecture mapping. Feature-level variability was examined in section 2.3.3 for all PL methods, as well as in the context of the generative programming techniques. The various approaches used today for product instantiation were examined in the context of FAD. Based on the conclusions drawn throughout the state of the art analysis, this section will define a plan for the achievement of a stronger feature-architecture mapping and consequently a more efficient feature-level variability and an improved generative product instantiation.

A Method for...

From the state of the art analysis, it became clear that a methodical approach is needed for the achievement of the aforementioned goals. This can well be served in the form of a method. The method must also seamlessly integrate with the established PL methodologies. In order for the method to be readily applicable, it must also support the technologies available at present, e.g. object-oriented programming languages, operating systems, development environments and tools.

The concrete characteristics of the method should be drawn from the positive aspects of the state of art approaches for the resolution of the feature-architecture mapping, feature-level variability and product instantiation. The method should also provide solutions to the problems identified in the state of the art methods regarding these issues.

...Stronger Feature-Architecture Mapping,...

From the conclusions on the methods addressing feature-architecture mapping it became clear that the derivation of architectural components cannot be efficiently based on use cases, like in FeatuRSEB. The examination of FAD has also shown that the utilization of the system external interfaces, problem domain entities and archetype instantiations based on the problem domain for the derivation of components has a positive effect on feature-architecture mapping. Nonetheless, FAD does not provide a consistent approach to enhance the mapping between features and the architecture. Finally, the use of solution domain entities for the creation of architectural components, e.g. FeatuRSEB control types and FAD archetype instantiations based on the solution domain illustrated that they are sometimes invaluable for the realization of a robust, maintainable system.

Consequently, in order to methodically enhance feature-architecture mapping, the components of the PL architecture must originate from the features themselves and the relations between them. Ideally, the application logic of one feature should be implemented in exactly one architectural component and the interface of that component should reflect the interaction of the feature with the other features of the PL. Nevertheless, this is not always possible. Solution domain entities conceived by the software architects of the system must also take part in the construction of the PL architecture. These conclusions should form the basis for the new method developed in this work.

Another important issue related to feature-architecture mapping is the actual mechanism that maps features to architectural components. FeatuRSEB makes excessive use of traces. This does not provide a strong mapping between features and the architecture, rather it adds extra effort for their creation and maintenance. Generative programming techniques utilize special constructs to achieve a mapping of features to the architecture. Again, this approach cannot be applied in all cases or for large systems and it also adds extra effort for the creation and maintenance of the introduced artifacts. Unfortunately, as mentioned above, it is not always possible to achieve a one to one relation between features and architectural components.

Therefore, the new method that is to support a stronger feature-architecture mapping must adopt some kind of traceability mechanism, which does not suffer from the identified state of the art problems.

...Efficient Feature-Level Variability...

The method developed in this work must also support the efficient application of variability mechanisms on the feature level. It has been shown that the contemporary variability mechanisms can be efficiently applied on the feature level only under certain circumstances. Some variability mechanisms can only be efficiently applied in the absence of feature tangling, while others are suitable only for small features. Yet other variability mechanisms require the adherence to certain architectures, which may not always suit the PL domain. Furthermore, every variability mechanism may jeopardize the system's conceptual integrity or require a large amount of effort for its application, depending on the degree of feature tangling. Generative programming techniques, as already discussed, can also be efficiently applied in a small scale and require the creation and maintenance of new artifacts. These issues must also flow into the new method.

Because the method should also support the use of contemporary technologies, it must make use of the available variability mechanisms. No extra constructs should be required for the application of the variability mechanisms, like e.g. hyperslices in the Hyperspace approach. Additionally, no extra architectural entities should be included into the system through the application of the variability mechanisms. This will ensure the preservation of the system's conceptual integrity.

...and Improved Generative Product Instantiation

The developed methodology must also resolve the issues identified in today's approach to product instantiation. From the various product instantiation models, the generator-based model was identified as the most efficient model for the generative development of PL products. Nonetheless, a number of disadvantages have also been identified, namely, the need for a mature domain, decreased evolvability due to the required evolution of configuration tools or DSL, limited number of products possible to be developed from the PL core, as well as an inefficient mapping of PL features to the final products.

The method developed in this work must provide solutions to the open issues of the generator-based model. It must be applicable to all domains, regardless the degree of experience already available in constructing systems in the domain. Extra tools used for the instantiation of the PL products must be insensitive to the evolution of the PL. The method should also exhaust the possibilities of the PL to produce valid products. Finally, the PL products must efficiently implement only the features selected by the PL customer.

The realization of the precise goals defined in this section will be presented in chapter 4, where the description of the methodology developed in this work is provided. Chapter 5 will evaluate the extend up to which the aforementioned goals are actually achieved.

2.7 Used Works

This section will present the works used in the developed methodology. This is done at this point to allow a fluent reading of the actual method sections, reducing the number of context switches for the reader. There are two works used in the developed methodology. These are the *profiles method* for the quantification of quality features and the use of search techniques for feature interactions.

The profiles method is developed in the QASAR method (sect. 2.4.1) to provide a quantitative definition of quality attributes for software systems. This method allows, through the definition of a set of usage scenarios, the functional assessment of quality attributes. A set of usage scenarios makes a profile. An example is the definition of a set of change scenarios for the maintainability of a system. The change scenarios describe concrete requirement changes that lead to changes in the software. These are measured with the line of codes that are affected by a change, in relation to the overall lines of code of the component that is affected by the change.

The profiles method is utilized in the method developed in this work for the quantitative specification of quality features. Similarly, for a quality feature like performance, a set of change scenarios is defined. These scenarios are then delegated to other functional features, which must satisfy them in a certain amount of time. Through this approach, the throughput or response time of a system can be assessed during development.

This work makes additionally use of interacts relations for the optimization of the system's architecture in terms of maintainability, variability and performance. Previous works on feature interaction, e.g. [CKM$^+$03] and [Gib97], define a feature interaction as a situation in which system behavior does not as a whole satisfy each of its component features individually. These works focus mainly on the telecommunications domain, where features have no knowledge of the existence of the features with which they interact. Although feature interaction is treated as a *uses* feature interaction in this work, where features do have knowledge of each other, previous works on feature interaction can be utilized for the identification of interactions between features.

The developed method makes use of such techniques for the identification of feature interactions, but provides other resolution approaches (sect. 4.5). An example of the techniques that can be used for the identification of feature interaction is the use of formal models [Gib97]. With this approach, an executable model of the system is developed to capture its dynamic behavior and a logical model, which defines the properties of the system that can be verified statically. The formal verification of the executable model is examined to assure that it fulfills the requirements of the static model. Although this approach might require a lot of project resources for its implementation, it may prove to be financially viable, e.g. in safety-critical domains.

Chapter 3

Case Study

This chapter's purpose is to provide an insight to the case study used in this work for the presentation of the proposed methodology for the enhancement of feature-architecture mapping. In fact, the method has been applied in a number of application domains, namely, in the domain of Integrated Development Environments (IDEs) [SRP04] and the Mobile Telecommunications domain [SRP06]. The IDEs domain provides an information-centric context, illustrating features like **Diagram Designer** and **Model-Code Synchronization**. The mobile domain on the other hand, covers a more limited, deterministic operating environment, with features like **MMS** and **Push**. The diversity of application domains is geared towards the maturation of the method itself, as well as towards a thorough evaluation of the method's attributes.

In this work, the method is applied to yet another application domain, namely, the artificial Neural Network (NN) domain. The NN domain is a computation-intensive domain, needing extensive algorithmic support and a well organized software architecture for large information processing tasks. This combination of characteristics makes it a challenging domain for the application of the proposed method.

In order to provide a background for further discussion of the key features of the system developed for this domain, section 3.1 will provide a basic introduction to the theory of NNs, while section 3.2 will discuss the actual case study in more detail.

3.1 Neural Network Theory

NNs have emerged from the effort to mimic the structure and operation of the human brain. The basic computational unit in the brain is the nerve cell or neuron (fig.

3.1). A neuron has:

- Dendrites (inputs)
- Cell body
- Axon (output)

A neuron receives input from other neurons, typically many thousands. Inputs are approximately summed. Once input exceeds a critical level, the neuron discharges an electrical pulse that travels from the body down the axon to the next neurons. The neuron is then said to have *fired.* The axon endings almost touch the dendrites or cell body of the next neuron. Transmission of an electrical signal from one neuron to the next is effected by neurotransmitters, chemicals which are released from the first neuron and which bind to receptors in the second. This link is called a synapse. The extent to which the signal from one neuron is passed on to the next depends on many factors, e.g. the amount of neurotransmitter available, the number and arrangement of receptors, the amount of neurotransmitter reabsorbed, etc..

One way brains learn is by altering the strengths of connections between neurons and by adding or deleting connections between neurons. Furthermore, they learn based on experience. The efficacy of a synapse can change as a result of experience, providing both memory and learning through long-term potentiation. One way this happens is through the release of more neurotransmitter.

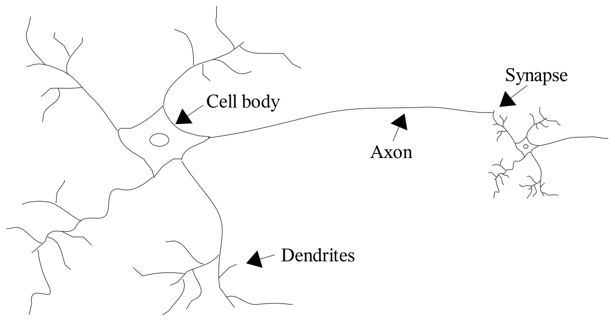


Figure 3.1: A simplified view of a biological neuron

An artificial neuron is a simplified model of the biological neuron (fig. 3.2). In an artificial neuron, the scalar input p is the analogy of the dendrites. The analogy of the axon is the neuron output a. The cell body is again the computational unit of an artificial neuron. As in the biological neuron, all inputs are weighted through the w scalar value and are added together. This sum of products is called the *weight function* of the neuron. Another scalar value, b, the so called *bias* is also added to the result. The output of the neuron is controlled by a function f, which is applied on the final result. This is the so called *transfer function* of the neuron and it simulates the firing of the biological neuron if a threshold is reached, e.g. if f was the hard-limit function giving 0 for $n < 0$ and 1 for $n \geq 0$, then an artificial neuron would have said to have fired if its output becomes 1. Many artificial neurons may also be combined in a layer, as shown at the right part of fig. 3.2. This layer has R inputs, each one connected to each of the S neurons of the layer. This technique allows for example the parallel processing of information by each neuron.

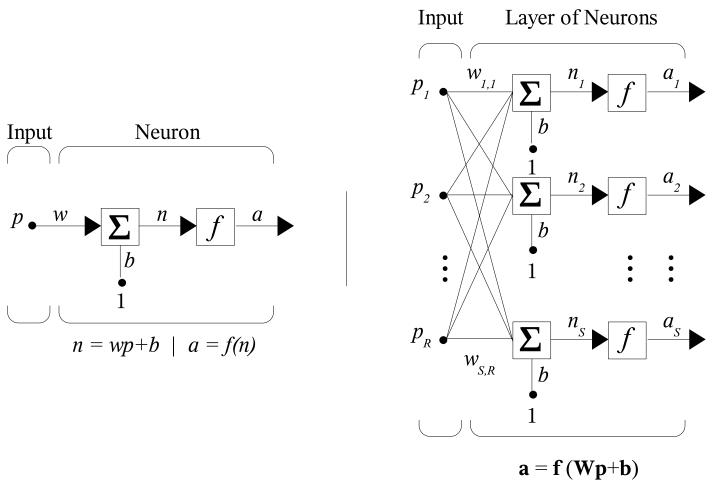


Figure 3.2: An artificial neuron and a layer of neurons

A NN is built from a large number of interconnected neurons, like in the human brain. A typical NN is shown in fig. 3.3. The NN consists of a series of neuron layers, where the output of the first layer serves as the input to the next layer. The output is the rightmost layer and it provides a vector of values, which depend on the previous layers' weights and transfer functions.

The central idea of NNs is to adjust the weights and biases, so that the NN exhibits some desired or interesting behavior. For this purpose, numerous training algorithms have been developed over the years. One family of training algorithms is the family of *supervised learning* algorithms. These algorithms receive a set of inputs, along with a set of corresponding outputs. These are the *training patterns*. Each input is successively applied to the NN and the NN output is supplied to a *performance function*, e.g. the mean square error (MSE) is calculated based on the given outputs. One such pass through the NN of all training patterns is called an *epoch*. After each epoch, the MSE is used in an algorithm to alter the weights and biases of the NN, so as to minimize the errors. Ideally, the MSE should reach zero, which indicates that the NN has "learned" for each input to provide the given output.

The structure of the neurons and the way they are connected define the *network architecture*. There exists a large number of network architectures and training algorithms for each architecture. A few examples are linear filters, backpropagation, radial basis and self-organizing maps. The provided information in this section have provided an overview of the key concepts of NNs for following the case study used in this work. For a more detailed discussion on NN theory, see [HDB96].

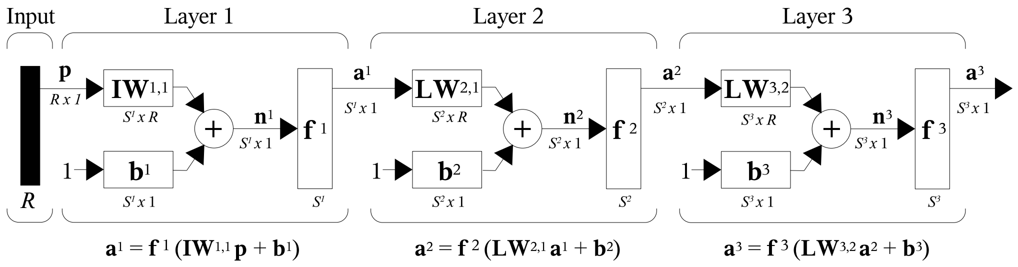


Figure 3.3: An artificial Neural Network

3.2 NN-Trainer PL

The case study used in this work involves the development of a Neural-Network Trainer Product Line (NN-Trainer PL). The NN-Trainer is a software for the instantiation and training of NNs. From the brief background information on NN theory it should have become clear that there exist many hard requirements for a NN-Trainer PL.

For the design of a NN, one needs to define a large amount of parameters for each network architecture (network topology, neuron distance, transfer function, etc.). Depending on these parameters, the NN must also be initialized, which again can be performed in a variety of ways. Furthermore, there exist a large number of training algorithms (Levenberg-Marquardt, Gradient Decent, Bayesian Regularization, etc.). Additionally, suitable performance functions must be available (Mean Square Error, Mean Average Error, etc.).

For certain network architectures, a training needs at least a set of training patterns. The latter must be somehow imported into the system in order to be used in the training process. In most cases, some kind of pre-processing of the training patterns is required. Depending on the size of the NN, the training may require very large amounts of computer memory, thus needing to be done in parallel, in a computer network. There is also need for the trained NN to be exported, so as to be integrated in other software applications.

In addition to all the above requirements, a NN-Trainer must also be efficient, robust and it must be easy to operate by providing visual aids and templates for the design of NNs, as well as feedback of the NN performance during training.

The author has been professionally involved with the development and extension of NN-Trainer systems in the Department of Modeling and Simulation of the University of Jena, Thueringen. The projects in which the author took part involved, among others, the use and extension of industrial NN tools, e.g. the MATLAB Neural Network Toolbox [TM06], as well as open source tools, like the Stuttgart Neural Network Simulator (SNNS) [oS06] and in-house developed NN-Trainer systems.

The NN-Trainer PL case study presented in this work models a complete environment for the design and training of NNs. The PL possesses more than 85 distinct features. Appendix A shows the initial FM of the NN-Trainer PL. The NNs developed with the resulting PL products can be applied to diverse problems, both in the research and industrial field. The development of the NN-Trainer PL has been exclusively performed with the method developed in this work. The implementation of the PL is based on the MATLAB Neural Network Toolbox. The basic functionality of MATLAB and the Neural Network Toolbox have been utilized as a platform for the implementation of various components for the NN-Trainer PL. The features developed in this case study are listed below.

- **Train-Start**
- **Train-Restart** & sub-features
- **Train-Stop** & sub-features
- **Train-Resume**
- **NN Periodic Save** & sub-features
- **NN-Activation**
- **Train-Statistic** - **Resources** & sub-features
- **Algorithm** - **Levenberg-Marquardt**
- **Train-Mode** - **Network**
- **NN-Export** - **Compiler** - **C++**

The developed components are written partially in the MATLAB programming language, but mainly in C++ in the Microsoft Visual Studio development environment.

The overall size of the components reaches approximately 12KLOC. The development effort was approximately 1 man-year.

Products of the NN-Trainer software are currently used in the University of Jena for the parallel training of very large feedforward backpropagation NNs for the solution of diverse problems. More precisely, one of the PL products for the training of large NNs in a computer network is used for the resolution of an inverse electromagnetic problem for the medical domain [Inn06].

It can be concluded that the NN-Trainer PL case study is suitable for the application of the proposed method. The system illustrates significant variability, both in the design and training process of a NN. The NN-Trainer case study reaches a level of complexity so as to be challenging, but does not grow over the available resources of this work. The case study has been driven by the quality requirements posed in an industrial environment, e.g. for the medical domain. Products of the NN-Trainer PL have been tested in a real-world context. This fact provided very useful feedback during the development of the proposed method and contributed significantly to its maturation.

Chapter 4

The Feature-Architecture Mapping Method

The solution for a stronger feature-architecture mapping is provided by means of the new **F**eature-**Ar**chitecture **M**apping (**FArM**) method. As identified in section 2.6, features should pose the driving force for the development of a PL. Therefore, the new method must set as its primary goal the derivation of architectural components from the PL features. Ideally, a one to one relation should exist between features and components. In real life though, this is not always feasible. This is mainly due to feature scattering and tangling. More precisely, the implementation of a feature's application logic is scattered throughout numerous architectural components. Moreover, the implementation of many features can be tangled within one architectural component. This phenomenon may occur due to the inherent complexity of features or their mere scattered nature, e.g. in the case of quality features. Nonetheless, the phenomenon of feature scattering and tangling mainly has its roots in the non-feature-centric approach to system design. A solution to this problem would consequently be a balanced mix between components relating directly to customer features and components that are derived from a pure architectural perspective. This set of components does indeed allow a one to one mapping to a PL FM. More precisely, to a *transformed* FM.

4.1 Overview

FArM receives as input an initial FM and produces as output the system's software architecture. More precisely, a transformed FM is developed as well as one architectural component for each feature of the transformed FM. These components are also mapped to a concrete software architecture, e.g. through the use of architectural styles (sect. 4.6). Each of the produced components implements the application logic of one feature and their communication reflects the feature interaction. The features of the initial FM are linked to components through a finite number of *traceability links*. Traceability links contain the rationale for any transformations that occurred to a feature. Consequently, one can select an initial feature and trace it down to one component or at most to a few components. Goal of FArM is to achieve a one to one relation between features of the initial FM and an architectural component. Although this is not always feasible, in order for FArM to approach this goal as much as possible, only the absolutely necessary transformations are performed on a feature. That is, features are only transformed, if the transformation considerably improves the system's maintainability and does not break the system's conceptual integrity. Nonetheless, the features of the transformed FM have a one to one relation to the architectural components. Therefore, FArM achieves a direct mapping for all features of the transformed FM and also a direct mapping for a large number of the initial features. In general, a stronger mapping is achieved between features and the architecture in comparison to the state of the art methods.

Prerequisites

The initial FM may be constructed with any domain analysis method available to the developers, as long as it is confined to the scoped PL domain and defines the requires and excludes relations between the features. Note that the method makes no assumptions of the features' nature or their hierarchy relations. The initial features may be of any kind, e.g. quality or functional features or they may represent special characteristics of the system, e.g. supported hardware or operating systems. Additionally, the hierarchy of the initial FM does not need to conform to any norms, i.e. features may have super-feature o sub-feature relations or they may have no hierarchy at all, e.g. can be provided simply as a list. For representative examples of the specification of a few features identified in the NN-Trainer case-study refer to Appendix B.

A FArM prerequisite is the requires and excludes relations between the features. As mentioned in section 2.1, these relations are used during the instantiation of PL products for the selection of features. Because the ultimate goal is to define architectural components that also reflect feature interactions, it is of vital importance to know which features require others and which features may not coexist in a product. These relations have a direct influence on the component interfaces. These minimal requirements placed upon the initial FM give great versatility to the users of FArM regarding the selection of the domain analysis method and thus, broaden the applicability of the FArM method itself.

Phases

The transition from the initial FM to the final transformed FM is an iterative process consisting of four phases, where the first three phases are distinct transformations as shown in fig. 4.1. The developers begin with transformation 1 (Trans. 1) and iteratively progress to the next transformations building a small part of the system's architecture in each iteration. If during a later transformation an omission or inconsistency in a previous transformation is detected, the developers return to that transformation, fix the problem and again work their way through the rest of the transformations to the point where they were before. This iterative approach of FArM is consistent to the state of the art development methodologies, e.g. to the Rational Unified Process or to spiral models of development, which combine the benefits of a waterfall and a pure iterative approach.

The first FArM transformation handles *Non-Architecture-Related* (NAR) and *Quality* features. Goal of this transformation is to resolve any non-functional features, thus producing a FM with features whose responsibilities can be expressed as some sort of function and are thus implementable.

The second transformation handles *Architectural Requirements*. Goal of this transformation is to add new functional features or extend existing features to satisfy requirements not visible from the customer perspective, rather from the architect's point of view. This allows the enhancement of the FM to include aspects of the system important to the system architects. This transformation also contributes to the aforementioned balanced mix between the customer and architectural perspectives.

These first two transformations deal with the identification of most of the features needed to implement an architecture that tightly maps to the PL features. For each of these features a respective architectural component has been developed, whose specification matches the one of the feature it is implementing. The next transformation builds upon these components by defining and optimizing their interfaces.

The third transformation identifies and resolves feature interactions. The identification of feature interactions is based both on the domain specific feature communication needs, as well as the FM hierarchy structure. The identified feature interactions

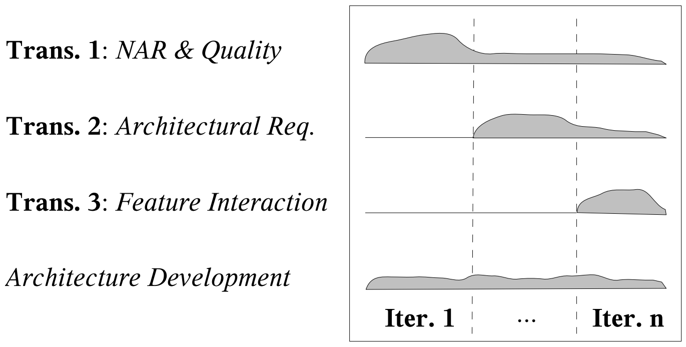

Trans. 1: *NAR & Quality*
Trans. 2: *Architectural Req.*
Trans. 3: *Feature Interaction*
Architecture Development
Iter. 1
...
Iter. n

Figure 4.1: FArM phases

are then resolved and optimized. This transformation effectively contributes to the decoupling of the respective architectural components and the enhancement of the system maintainability. The optimization of the feature interactions has also a positive impact on the communication between the respective architectural components. It contributes to the encapsulation of components and the enhancement of the system's variability. Based on the various feature interactions, interfaces are derived for the respective architectural components.

When the developers reach the architecture development phase, the system components have been derived along with their interfaces. The developers, if they haven't already done so in previous iterations, should decide for a specific PL architecture. For example, they may decide to make use of either the Layers, Microkernel or Blackboard architectural style for the PL and place each of the derived components within this architectural context. This process will effectively add more architecture related interfaces to some of the components, e.g. if the Microkernel architectural style is chosen, the components should receive methods for dynamic loading and termination. Finally, during the implementation of the components new interfaces may be added or omissions may be identified, which in turn lead back to one of the transformations.

Note that the FMs produced after each transformation are not discarded after the completion of the FArM development process. On the contrary, they are preserved as a documentation of the various transformations of the initial FM and as a mapping between features of the initial FM and the software architecture. This is achieved through the various traceability links created through the application of the elementary transformations on features as shown e.g. in fig. 4.2.

Organization

The application of the FArM method is facilitated within an organizational context. More precisely, the aforementioned "FArM developers" must be a compound team of feature analysts and system architects. The former have profound knowledge of the domain analysis techniques and preferably also of the domain itself. Their responsibilities are the creation of the initial FM and their collaboration throughout the transformations in the FArM phases, mainly on the more abstract feature level. The system architects on the other hand are concerned with the component development and the PL architecture. They work primarily on the architecture level. Nonetheless, the cooperation of these professionals is of great importance. They must be able to communicate the information gained from the feature or architectural level to the other party. This collaboration will provide vital feedback for the decision-making process regarding system maintainability, variability and the system's conceptual integrity. The roles of each party will be noted where appropriate during the detailed discussion of the intricacies of the FArM phases in the upcoming sections.

4.2 Elementary Transformations

Before delving into the details of the FArM transformation phases, a discussion of the transformations that may be directly performed on a feature that is provided. In each of the FArM transformations, only a small number of *elementary transformations* is allowed to be performed on any feature. These are shown in table 4.1. The FArM elementary transformations are an instrument in the hands of the developers to promote a structured development approach throughout the transformation phases. In each phase, the elementary transformations are allowed to be performed only to features of specific nature and always in the same manner. This assists the developers significantly in the decision-making process during the transformation phases. It enforces the proper handling of features, protecting them from erroneous transformations that may, e.g. completely remove a feature, which is valuable to a system stakeholder.

The simplest of all elementary transformations that may be performed on a feature is the *Direct* transformation. If a feature needs to be transformed and that feature has no influence on other features, then it is directly removed from the FM and instead, a traceability link is added between that feature and the root feature of the transformed FM, as shown in fig. 4.2. Examples of such features are hardware-

Elementary Transformations	
Transformation	Description
Direct	A feature is directly removed from the FM without influencing other features and a traceability link is created between the feature and the root feature of the transformed FM
Merge	A feature's specification is merged with the specification of another feature and a traceability link is created between the two features in the transformed FM
Create	A feature is transformed through the creation of new features to implement the feature's specification. Traceability links are created between the feature and each of the newly created features in the transformed FM

Table 4.1: FArM Elementary Transformations

related features that have no influence on the software architecture. This elementary transformation guards against the complete removal of a feature without leaving any trace in the system. Each feature in the initial FM is important to a PL stakeholder and therefore, its removal from the system must be treated with caution. The addition of the traceability link in this elementary transformation saves at least the rationale behind the removal of the feature and thus allows the indirect involvement of the feature in the system development process.

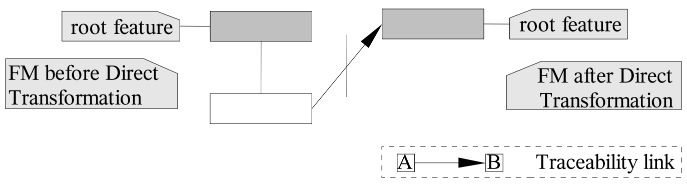

root feature
FM before Direct Transformation
root feature
FM after Direct Transformation
A
B
Traceability link

Figure 4.2: Direct Elementary Transformation

The *Merge* elementary transformation is applied to features that cannot be implemented as is and could or must be integrated with other features. Representative examples are quality features. In such cases, the feature specification is merged with the specification of other features and traceability links are added to each one in the transformed FM (fig. 4.3). This elementary transformation guards against redundancy during component derivation. If a feature cannot be implemented directly or it influences a large part of the system, then it is initially checked if it can be merged with other functional features that perform similar operations and could logically be extended to support some other operation. This limits the number of components effectively derived and thus guards against an explosion of the number of features/components in the system. Of course, such transformation must also be used with caution, so as not to break the system's conceptual integrity, e.g. a feature should not be merged with another if this means that the other feature would need to perform an operation unrelated to its current functionality.

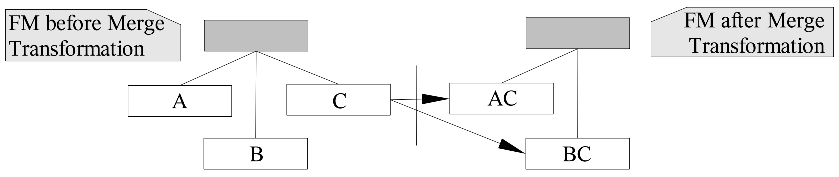


Figure 4.3: Merge Elementary Transformation

The *Create* elementary transformation handles the instantiation of new features based on the transformed feature. This transformation may be triggered for a number of reasons. It may be applied when a feature is identified to have unrelated or too many responsibilities. Additionally, there may exist other requirements on the system, not identified during the domain analysis, which need to be added to the system in the from of features, e.g. architectural requirements. Finally, it may be the case, that the aforementioned transformations do not suffice for a feature's complete transformation. In each of these circumstances, new features are created to satisfy the specification of the transformed feature and traceability links are added between them (fig. 4.4). When applied properly, this transformation contributes to the enhancement of the system's maintainability, since it handles inconsistencies in the derived component responsibilities, thus enhancing their encapsulation. It also allows the integration of non-customer related system requirements into the system design and handles them on the feature level. This contributes additionally to a balanced mix of customer and architectural features.

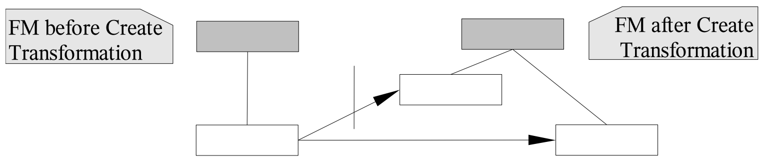

FM before Create Transformation
FM after Create Transformation

Figure 4.4: Create Elementary Transformation

There exist a few general rules for the applications of the elementary transformations. At first, one or more of the elementary transformations can be applied for the transformation of a feature, i.e. the specification of a feature may be divided into parts and each of these parts is then transformed using a different elementary transformation. Furthermore, a feature specification must be considered for transformation with each of the elementary transformations in the aforementioned order, i.e. the specification should be initially examined if it can be directly transformed, then merged in other features and finally, if new features can be created. Note, that these three transformations together cover all possible scenarios for the transformation of a feature. Detailed examples of each of the elementary transformations and their application on specific features are given throughout the discussion of the FArM phases in the following sections.

4.3 NAR & Quality Features

This first transformation is concerned with the resolution of all Non-Architecture-Related (NAR) and Quality features. Goal of this transformation is to create a FM where only functional features are present. This will increase the likelihood of a direct implementation of each transformed feature in an architectural component, since they would then represent functional attributes of the system.

4.3.1 Non-Architecture-Related (NAR) Features

FArM strives to remain independent of the domain analysis method used. This broadens the applicability of the FArM method. For this reason, no assumptions are made regarding the nature of the features present in the initial FM. Therefore, it may be the case that features having no impact on the software architecture of the system are present in the initial FM. These features are referred to in FArM as *Non-Architecture-Related* or for short, *NAR* features. Goal of this transformation is to identify such features and "resolve" them. With the term "resolve" it is meant that these features are transformed in FArM in order to be satisfied and effectively removed from the FM.

NAR features, as mentioned earlier, are features having no direct impact on the *software* architecture of the PL. Arguably, every feature has some effect on the architecture of a system regardless if it is e.g. a purely hardware-related feature. FArM though is concerned with the software architecture of the PL. Therefore, a NAR feature is such a feature that has a minimal impact on the software of the PL system, although it might have a larger impact on another aspect of the system.

A rule of thumb to determine if a feature has a direct influence on the software architecture is to ask the question: "If a component was to implement that feature, which responsibilities would this component have?". If the answer to this question cannot be unambiguously given, then it is most likely that this feature is a NAR feature. Because of the minimal effect that NAR features have on the software architecture of the PL, they are entirely transformed with the direct elementary transformation (sect. 4.2). NAR features are thus removed from the FM and traceability links are added between the features and the root feature of the transformed FM to hold the transformation decisions and rationale.

In addition to the above rule of thumb, FArM identifies three categories of NAR features as shown in table 4.2. The categories *physical*, *external* and *business* cover practically all NAR features encountered throughout the various case studies where FArM has been applied. The following sections will take a closer look at each of these categories.

Non-Architecture-Related Categories	
Category	Description
Physical	Features related to hardware aspects of the system
External	Features related to services or resources used by the PL software
Business	Features related to business aspects of the system

Table 4.2: NAR feature categories

Physical NAR Features

Physical NAR features are the features that represent a hardware part of the system. Such features are often important to PL customers and are encountered frequently in the domain analysis process and thus in the initial FM. Physical NAR features are identified depending on their relation to hardware and the impact this hardware has on the software architecture of the system. Physical NAR features should be completely transformed with the direct elementary transformation (sect. 4.2) alone. If this is not possible, then the developers should reconsider the nature of the feature at hand, since this is an indicator that the feature may belong to another NAR category or even not be a NAR feature at all.

For the NN-Trainer case study (chapter 3), physical features were encountered in the **Hardware - CPU**, **RAM**, **Network Adapter** feature hierarchy (fig. 4.5). These features indicate the hardware needs a NN-Trainer system has. The **CPU** and **RAM** feature specifications contain the minimal CPU performance and amount of RAM needed, e.g. during the training of a NN. These two features are also mandatory, i.e. they must always be selected from the PL customer. During product instantiation, the available CPU and RAM of the customer's hardware platform, where the NN-Trainer system is to run, is checked against the minimal requirements defined by the **CPU** and **RAM** features respectively.

The optional **Network Adapter** feature indicates that the NN-Trainer system may need to have a network connection in order to operate properly. The feature also provides the requirements placed upon the network connection. The **Network Adapter** feature is present, because a NN-Trainer product may be licensed for use in a network environment. In this case, a network connection must be present for the product to boot and operate. For this scenario, the **Network Adapter** feature also defines the minimum speed the network connection should have, e.g. proper operation of the software is guarantied with a 56Kbps modem connection.

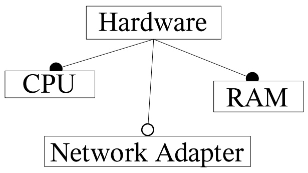


Figure 4.5: The **Hardware** feature hierarchy is composed of Physical NAR features

It is clear that these features are purely hardware related. In the NN-Trainer case study, the **CPU** and **RAM** features refer merely to the minimal requirements of the computer system upon the NN-Trainer is to run. The NN-Trainer system is designed to run on powerful PCs and therefore, CPU and RAM considerations have a small influence on the system's software architecture.

The identification of features as being physical NAR features may at first glance seem to be a simple process of just identifying if a feature represents hardware rather than software. In practice though, caution is required. As already mentioned, NAR features should have minimal or ideally, no impact on the software architecture. In practice, many physical aspects of a system may have enormous impact on the software architecture. This is especially true for embedded systems, where limited resources are available. This fact denotes the importance of a precise feature specification. The latter is a vital precondition for the identification of the proper nature of a feature by the PL developers.

For the NN-Trainer case study, the fact that the system is to operate on powerful PCs with the minimal requirement of a modem connection limits the impact that the available CPU, RAM and network adapter have on the system's software architecture. A counter-example that denotes the importance of hardware related features can be found in the application of FArM on the domain of mobile phones [SRP06]. In this case study, the **Battery** feature was examined. In the context of mobile phones, the **Battery** feature refers to the characteristics of the battery performance of the mobile phone product. Among others, the required battery-life duration is specified in this feature. During the NAR FArM transformation phase, one could identify the **Battery** feature at a first glance as being a physical NAR feature. This is false. The **Battery** feature in the context of mobile phones is a functional feature that needs to be partially implemented in an architectural component.

Mobile phones are embedded systems built to operate for long periods of time. Data transmission in a wireless network requires battery power, especially in case of a "weak" network connection. Additionally, battery duration is an important purchase criterion for many mobiles phone users. Therefore, the **Battery** feature specification imposes the application of hard algorithms on the entire software system for the optimization of battery performance. For example, unnecessary network traffic must be minimized, while connection policies and server-side software must contribute to the optimal management of bandwidth. In the mobile phone PL a `Battery` component is needed to manage network traffic, while parts of the **Battery** feature specification must also be implemented as server-side software components.

External Features

It is not seldom that features in an initial FM relate to services or resources that are not provided by the PL, rather by a third party. Such features are present because of completeness purposes or because of their influence on other features and the PL in general. These features are referred to as *External* features in FArM. Like in the case of physical NAR features, external features may have some influence on the overall system architecture, but still have no direct influence on its software architecture, i.e. they have no direct influence on the major software components and their interfaces. External features are primarily identified by the fact that they are not implemented in the PL itself, although they are present in the FM. External features must be completely transformed with the FArM direct elementary transformation.

In the NN-Trainer case study, the mandatory **OS - Windows** feature hierarchy (fig. 4.6) is identified to be composed of external features. The **OS** (Operating System) feature represents the software platform upon which the NN-Trainer is to operate. The NN-Trainer PL developers, in collaboration with the application engineers of the NN-Trainer system have early made the decision to support only the Windows platform. This decision is based on the fact that a stable API is provided for the various windows versions, as well as on the market segment covered by this operating system. The development costs for supporting other OS platforms and at the same time providing satisfying performance, e.g. for the UNIX, Linux and Macs OSs, would exceed the project's budget.

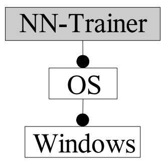


Figure 4.6: The **OS** feature hierarchy as External NAR features

The **OS - Windows** feature hierarchy is present in the initial FM to indicate the fact that the NN-Trainer system is available only for the Windows platform, i.e. for completeness purposes. Additionally, these features do indeed influence the general system architecture, but not directly its software architecture. The features are transformed with the use of managerial solutions, i.e. the decision to use the C++ .NET framework and partially the Windows API. Note that the use of these technologies does influence the software components, but on a lower implementation level. These technologies have neither a direct influence on the PL components' application logic, nor the interfaces derived from this logic.

The identification and transformation of external features also needs consideration. One should not confuse external features with features related to *external systems*. An example is the NN-Trainer PL feature hierarchy **NN-Export** - **Compiler** - **Java**, **C++**, etc. (fig. 4.7). This feature hierarchy indicates that the NN-Trainer system provides a function to export trained NNs. This is achieved through the generation of C++ or Java source code, which can be compiled by the respective compilers. These features allow the integration of trained NNs into C++ and Java applications.

The **NN-Export** feature hierarchy obviously depends on external services and resources provided by the C++ and Java compilers. Nonetheless, the source code for these compilers must be generated by the PL system. This is the responsibility of the aforementioned features. Therefore, these features might at first glance seem to be external features, but in reality they are functional features, each requiring an architectural component for its implementation.

Business Features

In the initial FM there may also be the case that a number of business-related features are present. These are primarily features that are of vital importance for the

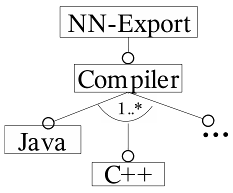


Figure 4.7: The **NN-Export** feature hierarchy

marketing and management stakeholders of the PL. Such features are referred to as *Business* features in FArM. As in the previous NAR categories, business features are directly transformed, since they have no direct influence on the PL software architecture. The primary transformations applied to business features are of managerial nature.

In the NN-Trainer case study the **Competitive Market Price** feature was present in the initial FM (fig. 4.8). This feature indicates the fact that the NN-Trainer products must provide a better performance to cost ratio, compared with other similar products of the competitors. This is a critical feature for the financial viability of the PL. This feature has been added from the marketing department of the PL. Such a feature has a wide impact on the overall PL system. It influences the size of the investment made and the amount of available resources, ranging from the number and qualifications of the developer team, up to the tools that may be used for system development. Nonetheless, for the NN-Trainer PL the system software architecture is not directly influenced by the **Competitive Market Price** feature.

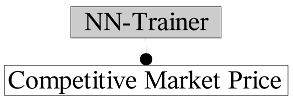


Figure 4.8: The **Competitive Market Price** feature

As already mentioned, business features are transformed mainly through the use of managerial solutions. For the NN-Trainer **Competitive Market Price** feature, the decisions are made to perform periodical risk analysis and to add a small number of experienced developers to the NN developer team. The periodical risk analysis will have the effect of the early identification of erroneous development decisions and their timely resolution. This practice guards against problems occurring at the beginning of development, when the cost of correction is still small compared to a later point in time. Acquiring experienced developers saves a large amount of development time, since they contribute to the efficient resolution of problems early in the development cycle. The latter also reduces the development costs and consequently the final market price of the PL product. These actions effectively allow the direct transformation of the **Competitive Market Price** feature and its removal from the FM after the first transformation phase. As already described in sect. 4.2, a traceability link between the feature and the root feature of the transformed FM is added, which also contains the previously mentioned actions taken for the feature transformation, as well as the underlying rationale of the transformation.

4.3.2 Quality Features

The quality attributes of a software system are of great importance for its stakeholders. If a system satisfies the functional requirements placed upon it, but does not also satisfy the quality requirements related to this functionality, then it is considered to have failed its purpose. Additionally, the quality attributes of a system can be easily understood by most of the system stakeholders, e.g. by customers, managers or the marketing department. Because of these two characteristics, quality features are almost certainly present in initial FMs.

Quality features have normally a broad impact on the software system. Therefore, they cannot be implemented directly in an architectural component. In order to achieve a direct mapping between features and the architecture, quality features should be appropriately handled. In FArM, quality features are identified and resolved. Like in the case of NAR features, the resolution of quality features involves the satisfaction of their specification and their removal from the initial FM.

The versatility of FArM regarding the preconditions placed upon the initial FM is also evident in the identification of quality features. FArM does not impose any limitations on the identification of quality features. This can be based on any standard available to the PL developers. For example, the quality views and models defined in the ISO 9126 standard [ISO01] can be readily used with FArM. Nonetheless, quality features must be quantitatively defined, i.e. their specification must clearly indicate in what ways quality features influence the functionality of the software. This does not mean that the quality features' specification must also provide a functional solution for their application, rather the constraints placed upon the functionality of the system. If no such quantitative specifications are provided for quality features, then they must be created at this point. The FArM developers may use the *profiles method*, as described in the QASAR method (sect. 2.7).

The resolution of quality features must take place through the combination of the FArM elementary transformations (sect. 4.2). More precisely, the specification of each quality feature is broken down into parts, upon which the elementary transformations can be applied. The FArM elementary transformations are applied in the order given in table 4.1. A part of the specification of a quality feature is initially considered for direct transformation. If this is not possible, then the developers consider the possibility of merging this part of the specification into pre-existing functional features. If these transformations fail, then new features should be created to satisfy this part of the quality feature's specification.

This iterative process for the transformation of quality features allows the resolution of quality features on the feature level. The consistent work on the feature level enhances the system's conceptual integrity, since all derived components originate from the FM. For instance, the utilization of design patterns for the resolution of quality features does provide adequate solutions, but also causes the addition of architectural components, which may not directly relate to features. This in turn diminishes the conceptual integrity of the software system.

There is actually a resemblance between the utilization of design patterns and the FArM *create* elementary transformation, which is used to create new functional features. Both approaches introduce new architectural components in the system originating from a quality feature. The difference between the two approaches lays on the fact that the FArM created components directly relate to features that are important to the system architects and have been approved by the feature analysts. In the case of design patterns, the new architectural components are exclusively related to the architectural perspective and have no relation to any comprehensible feature. This deteriorates the conceptual integrity of the system.

Recoverability

In the NN-Trainer case study the developers utilized the ISO 9126 standard for the identification of quality features. Consequently, the system was examined from the perspective of reliability. This refers to the probability with which software will not cause the failure of a system for a specified time under specified conditions. This probability is a function of the inputs to and use of the system, as well as a function of the existence of faults in the software. From the reliability quality view the developers concluded that the capability of the NN-Trainer system to re-establish a specified level of performance and recover the data directly affected in the case of a failure, is of vital importance. This is especially the case during the training of large NNs for long periods of time, where a failure of the software and inability to recover from the failure would lead to very high costs. For this reason, the developers added the **Recoverability** feature in the initial FM (fig. 4.9).

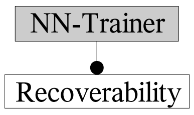


Figure 4.9: The **Recoverability** quality feature

The specification of the **Recoverability** feature is given below (see also Appendix B):

"The NN-Trainer system shall periodically save the state of a NN during its training. The amount of time between the saving process shall be flexible and shall be determined by the user upon the initialization of the training. A NN training shall be able to be resumed from the saved training state..."

It is clear that the feature's specification is indeed quantitatively defined, as required by the FArM method. The specification clearly states the influence of the **Recoverability** feature on the functional aspects of the system, namely, the saving of a NN's state during training and the ability to resume a training from that state. Note also that the feature's specification does not provide any concrete functional solutions for the implementation of the feature, rather it merely defines the influence this quality feature has on the NN training process. Since the **Recoverability** feature satisfies the FArM requirement of being quantitatively defined, the developers may proceed with its resolution.

The aforementioned feature specification is broken down into parts upon which the FArM elementary transformations are applicable. The specification parts, the elementary transformations used and the related features are shown in table 4.3. The **Recoverability** feature specification is broken down into two parts, the one referring to the saving of a NN's training state and the one referring to the ability to resume a NN training. For the first part of the specification referring to the saving of a NN's training state, both the *merge* and *create* FArM elementary transformations are applied. The merge transformation affects the **Train Start** feature and the create transformation causes the addition of the new **NN Periodic Save** feature. For the part of the specification relating to resuming a NN training from a saved state the *create* elementary transformation is used and the new **Train Resume** feature is added to the FM.

The **Train Start** feature has already been identified during the domain analysis process and is therefore present in the initial FM (left part of fig. 4.10). The feature indicates the basic ability of the user to start the training of a NN. This

Recoverability Transformation		
Transformation	Feature	Specification Part
merge	**Train Start**	*"The NN-Trainer system shall periodically save the state of a NN during its training. The amount of time between the saving process shall be flexible and shall be determined by the user upon the initialization of the training"*
create	**NN Periodic Save**	
create	**Train Resume**	*"A NN training shall be able to be resumed from the saved training state"*

Table 4.3: Transformation of the **Recoverability** quality feature

feature is directly affected by the first part of the **Recoverability** quality feature's specification referring to resuming a NN training. The **Train Start** feature handles the initialization of a NN training and consequently also takes part in resuming a NN training. The specification of the feature is extended to indicate also that the transformed **Train Start** feature must be able to start a training for a new NN and for a partially trained NN, effectively resuming its training. A traceability link is added between the initial FM and the transformed FM to hold the aforementioned rationale. Eventually, the resulting `Train Start` architectural component will be responsible for starting the training of a new NN and triggering the resuming a NN training.

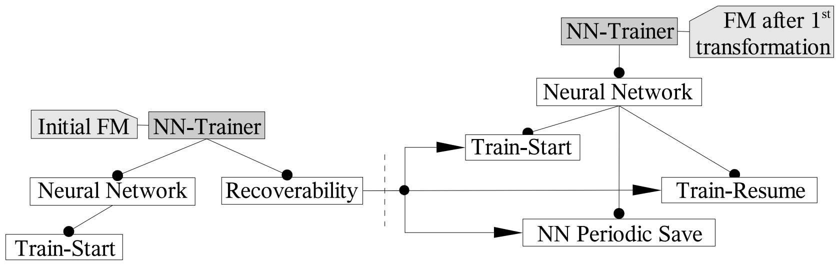


Figure 4.10: The **Recoverability** quality feature transformation with traceability links

The **Train Start** feature has a clear responsibility and is important to the PL stakeholders. It is therefore not possible to extend it in such a way that it can completely satisfy the first part of the **Recoverability** feature's specification. This would lead practically to a considerably altered feature, compared to the **Train Start** feature of the initial FM. This guided the developers to create a new functional feature. The new feature is given the name **NN Periodic Save**. The specification of this new feature is identical to the part of the **Recoverability** feature's specification. A traceability link is added between the two FMs (fig. 4.10) and a new architectural component is created for the implementation of the **NN Periodic Save** feature.

The second part of the **Recoverability** feature's specification shown in table 4.3 is resolved through the creation of a new functional feature. The newly created **Train Resume** feature is responsible for taking the proper steps and actually resuming a NN training from a previously saved state. It is clear that the respective component must communicate with the `NN Periodic Save` component to achieve this task. The whole process will be initiated from the `Train Start` component. The combination of these features completely satisfies the **Recoverability** feature, which is no longer present in the transformed FM.

Efficiency

Based on the ISO 9126 standard, the **Efficiency** quality feature was identified during the domain analysis process for the NN-Trainer PL (fig. 4.11). The **Efficiency** quality feature refers to the capability of the software product to provide appropriate performance, relative to the amount of resources used under stated conditions. This feature's specification is given as follows:

"The NN-Trainer PL shall provide appropriate response, processing times and throughput rates, while using certain amounts and types of resources for each of its functions under stated conditions..."

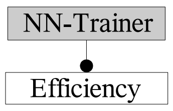


Figure 4.11: The **Efficiency** quality feature

It is clear from the above definition that this feature is not quantitatively defined, which is a vital precondition for its transformation in FArM. For this reason, the QASAR *profiles* method can be used as described in section 2.7. For this purpose, the **Efficiency** quality feature's specification is broken down into two parts. The first part refers to the *time behavior* of the NN-Trainer system and the second part to its *resource utilization*. For these parts, the developers define a set of usage scenarios with concrete values in each case. A sample of the usage scenarios for the **Efficiency** feature's specification is shown in table 4.4.

After the feature is quantitatively defined, the developers must apply the FArM elementary transformations in the order given, namely, first examine if the parts of the feature's specification can be directly resolved, if they can be merged with existing functional features and finally, if new functional features must be created to satisfy the specification. In the **Efficiency** feature's specification, the *time behavior*

Efficiency Scenarios	
Specification	Scenario
Time Behavior	*"The instantiation of a NN with less than 10000 parameters shall not last more than 5ms"*
	"The time required for one training epoch of a NN with less than 10000 parameters with the Levenberg-Marquardt algorithm shall not last mor than 5s"
Resource Utilization	*"The size of a NN with 10000 parameters shall not exceed 15KB"*
	"The training of a NN with less than 10000 parameters shall not require more than 50MB of memory"

Table 4.4: Scenarios for the **Efficiency** quality feature specification

part can be directly resolved through the use of a compiled programming language, instead of an interpreted language. The developers decide to use the C++ compiler for the development of the system, rather than the Java programming language. This solution does not directly affect the high level software architecture of the system and also none of the features of the initial FM, although this decision has a large impact on the implementation details of the PL.

The direct resolution for the time behavior part of the feature's specification does not provide its complete resolution. The **Efficiency** scenarios are then examined individually for their resolution with the *merge* and *create* FArM elementary transformations.

The first scenario is satisfied with the merge elementary transformation. The specification of each feature taking part in the instantiation process of a NN is extended to include the limitations posed by the efficiency scenario. For example, the **Initialization** feature (fig. 4.12), which is responsible for providing initial values to the weights and biases of a NN is extended, so as to operate within the given time limits. The resulting `Initialization` component must perform the initialization of a NN with less than 10000 parameters (weights+biases) within 1ms. This limitation has an impact on the algorithms used for the initialization and their implementation within the component.

The second scenario is resolved with the merge elementary transformation. In this case, all features taking part in the training of a NN are affected. For instance, the **Levenberg-Marquardt** (fig. 4.12) feature's specification is extended so as to impose the training of a NN within 5ms per epoch.

The third scenario refers to the resource utilization required of the NN-Trainer PL. Again each feature related to the instantiation of a NN must be adapted to the needed size limits. An example can be found in the **Topology** feature (fig. 4.12).

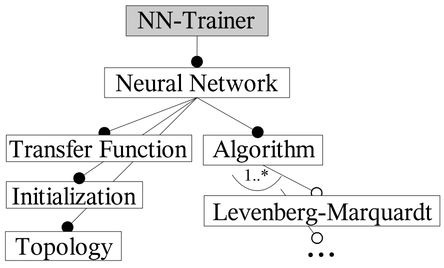


Figure 4.12: Partial view of the features involved in the resolution of the **Efficiency** quality feature

This feature defines the position of each neuron of a NN, as well as the overall NN architecture. This information in the resulting component must not exceed, e.g. 5% of the allowed 15KB. For instance, a proprietary binary format for saving this information could be used within the component.

The last scenario is also satisfied with the merge elementary transformation. More precisely, the features related to the training of a NN are extended, this time to retain the amount of memory under the 50MB limit. One of the features affected is the **Transfer Function** feature (fig. 4.12). This feature is responsible for the algorithms used during the activation and training of a NN (capt. 3). The feature's specification is extended to illustrate the fact that the feature will not use more than 5MB of memory for the various variables needed for the application of the transfer function.

Note that no new functional features were needed for the **Efficiency** feature's transformation. The direct and merge elementary transformations suffice for its complete resolution. The feature's specification is partially directly transformed and the rest of its specification, namely, the usage scenarios, are merged with the related PL functional features.

4.4 Architectural Requirements

Ideally, at this point of the overall FArM method all quality and non-architecture-related feature should have been effectively resolved. This implies that the transformed FM after the first transformation phase contains only functional features. Furthermore, for each of these functional features an architectural component has been derived having a specification identical to the one of the feature whose application logic it is implementing.

In order to obtain a strong mapping between features and the architecture, ideally, all components derived at this stage of FArM should also compose the final PL architecture. Unfortunately, in practice this is not always feasible. There may exist pure architectural aspects of the system that need to be taken into consideration for the development of a robust and maintainable architecture. This thesis is also supported by most of the state of the art PL methods, e.g. in the case of the FAD solution domain entities (sect. 2.4.7). More precisely, it is sometimes unavoidable to introduce entities in a software architecture that are related directly to the architectural requirements of a system. These entities play in most cases a crucial role in the maintainability and flexibility of the system.

Therefore, a compromise between a purely feature-oriented and a solution-domain-oriented architecture must be achieved. FArM strives to achieve a balance between these two views in this second transformation phase based on *Architectural Requirements.*

In the second FArM transformation phase (fig. 4.1) the architectural requirements of the PL system are handled. Like in the case of the quality features' resolution, the architectural requirements of the PL system are gathered from the PL developers and their specification is examined based on the elementary FArM transformations. Consequently, some parts of the specifications are directly resolved, others are merged with the specifications of pre-existing functional features, while others are satisfied through the creation of new functional features. It may also be the case that a combination of elementary transformations is applied to the same specification part, as was the case for the **Efficiency** quality feature (sect. 4.3.2).

This similar methodical approach promotes the consistency of the FArM method. Furthermore, FArM supports the resolution of architectural requirements either in pre-existing or in new functional features. In the former case, the architectural requirements become an integral part of a feature, without considerably changing its original purpose. In the latter case, new functional features are created, which are comprehensible from the system architects, but have also been approved from the feature analysts and should therefore be also comprehensible by most of the PL stakeholders. This fact enhances the mapping between features and the architecture, since the derived architectural components are tightly related to aspects valuable to the system stakeholders, while at the same time they satisfy important architectural aspects of the system.

Matrix Library

One of the architectural requirements placed upon the NN-Trainer PL is the development of a library for matrix operations. The system architects identified the intense matrix manipulation operations required for the implementation of many parts of the NN-Trainer system, e.g. for the creation of training and validation patterns or for the training of the NNs themselves.

Like in the first transformation phase, the FArM elementary transformations are applied sequentially to this architectural requirement, i.e. first the requirement is considered for direct transformation, then for merging with existing functional features and finally the creation of new functional features is considered.

In the case of the *Matrix Library* architectural requirement the FArM direct resolution is solely needed to be applied. For this architectural requirement the decision was made to purchase a third-party library for matrix operations. The decision was based on the effort and cost needed to write a high performance matrix manipulation library, compared to the cost of purchasing an existing one. The third-party library is purchased along with its source code for the case of customizations. In the NN-Trainer case study, the Matrix TCL Pro library was used [Ltd06].

External License Manager

Another architectural requirement placed upon the NN-Trainer PL is the use of an external license management technology for the licensing of NN-Trainer products. This architectural requirement expresses the need to use a third party solution for the implementation of licensing in the NN-Trainer system. This need arises for a variety of reasons. On the one hand, the use of a third party solution leads to the reduction of the development costs of the system. On the other hand, it also increases the quality of the product, since the used technology must respectively fulfill high quality standards and is thoroughly tested in an industrial environment. This architectural requirement defines additionally that the FLEXlm license manager [Mac05] must be supported in the first version of the NN-Trainer system, but it should also be possible to switch to another external license manager product with relatively small effort.

This architectural requirement is resolved through the use of the FArM elementary transformations. At the beginning, the specification of the architectural requirement is examined for direct resolution, i.e. if it can be satisfied without influencing any features of the current transformed FM. The developers realized that no direct resolution of this architectural requirement is possible. For example, the requirement specification cannot be resolved through the use of managerial or organizational solutions.

Since the architectural requirement cannot be directly resolved, the developers proceed to the resolution of the requirement by merging it with existing functional features. For this elementary transformation the developers identified the **License** feature, which is present in the transformed FM (fig. 4.13). The **License** feature was already part of the initial FM and was not affected by the first FArM transformation phase. This feature specifies the licensing scheme required by the owners and marketing department of the PL. More precisely, it indicates that *each* product of the PL must be licensed. This can be seen from the mandatory nature of the **License** feature. Additionally, the internal specification of the feature dictates that a customer of the PL must register his/her product based either on a single, group or network license scheme. This includes e.g. key-generation, implementation of encryption algorithms, license file management, etc. For a more formal specification of the **Licence** feature refer to Appendix B.

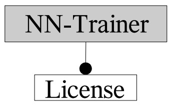


Figure 4.13: The **License** feature used for the resolution of the External License Manager architectural requirement

Through the FArM elementary transformation, the specification of the architectural requirement is merged with the specification of the **License** pre-existing functional feature. The specification of the **License** feature has indicated until this point in time that the feature should provide a licensing mechanism for the NN-Trainer system. This responsibility is now delegated to the external license manager and the **License** feature specification is altered. The feature now refers to the encapsulation of the NN-Trainer system from the external license manager software. From an architectural perspective, the component which is derived from the **License** feature will have the responsibility of providing a layer between the NN-Trainer system and the external license manager, thus providing the desired flexibility. This can be shown in the case of the FLEXlm license manager.

The FLEXlm license manager is based, as most license manager software, on an SDK (Software Development Kit) and a set of tools to impose a licensing scheme. The SDK provides a simple way of integrating code constructs into the source code of the software to be license-protected. This is done in the case of the FLEXlm software through special method calls. These method calls are used to enclose code fragments implementing a feature that needs to be protected (listing 4.1). The FLEXlm method-calls handle the communication with a license server, which checks the access rights of a user to the protected feature. The tools provided by the license manager software include the actual license server, key-generation, the mapping of keys to features, as well as the management of licenses and users.

Listing 4.1: FLEXlm macros

```
#include"lmpolicy.h"

if(CHECKOUT(LM_RESTRICTIVE,"feature","1.0","license.dat")){
  PERROR("Checkout failed");
   exit(-1);
}

// Checkout succeeded
// Actual application code here

CHECKIN(); // Done with "feature", check it back in.
```

The responsibility of the component implementing the **License** feature is now the implementation of specific method-calls that are to replace the ones provided by the FLEXlm SDK. The `License` component would then internally place the actual calls to the FLEXlm software. This effectively protects the rest of the system from changing upon switching to another license manager and thus satisfies the posed architectural requirement.

Network Training

The software architects identified, based on their experience, that some training algorithms require a large amount of memory in order to operate. The amount of memory needed even for medium-sized NNs exceeds by far the RAM of a single PC. This fact led to the architectural requirement of training NNs within a network environment.

This architectural requirement is initially examined for direct resolution. During this examination, it becomes clear that a large amount of data must be transferred on the network environment upon which the NN-Trainer is to operate. For this reason it is decided that a broader network bandwidth is required for the proper operation of a network training. During the resolution of the **Hardware** feature hierarchy, described in section 4.3.1, it was identified that the system needed merely a modem connection to operate properly. This requirement was captured in the form of a traceability link between the **Network Adapter** feature and the **NN-Trainer** root feature of the transformed FM (sect. 4.2). This traceability link specification is effectively changed in this transformation phase to indicate the new requirements of the system. More precisely, the minimum network requirements of the system are now increased to at least a 56Mbps DSL connection. This constraint contributes partially to the realization of the training of NNs in a PC network.

Nonetheless, this architectural requirement cannot be completely resolved through the direct elementary transformation alone. The architects, in cooperation with the feature analysts, examine if there are existing functional features, which can be used for the actual implementation of the network training. This search unfortunately yields no features that are closely related to this architectural requirement and could thus take on the responsibility of implementing this architectural requirement.

It is clear at this point that a new feature must be added to the NN-Trainer FM for the resolution of the Network Training architectural requirement. The PL developers create the new **Train-Mode** feature hierarchy shown in fig. 4.14. The **Train-Mode** super-feature refers to the environment in which a training takes place. This can be either a single PC, indicated by the **PC** feature or on a **Network** environment. The marketing department made the decision to promote this functionality in separate, i.e. a customer can train a NN either on a single PC or on a PC network, which is indicated by the cardinality of `1`.

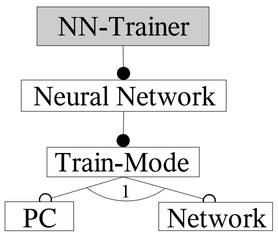


Figure 4.14: The **Train-Mode** feature hierarchy

The **Network** feature in the **Train-Mode** hierarchy has now the responsibility of performing the intensive operations needed by the algorithms, e.g. matrix inversions. Additionally, the derived component must perform the network communication and synchronization between the different NN-Trainer instances.

For example, if a customer selects the **Network** feature, then many instances of the

NN-Trainer can be started on different networked PCs. The NN-Trainer instance that initiates the training of a NN can then use the other NN-Trainer instances to invert a large matrix. The matrix inversion is performed by the `Network` component of each instance.

The component derived from the **PC** feature provides a simple version for the completion of the algorithm intensive operations, e.g. a local, one-PC matrix inversion. The component derived from the **Train-Mode** feature is responsible for providing an abstract interface for the algorithm intensive operations and for delegating the operations to the selected component. Additionally, the `Train-Mode` component is responsible for making sure that the operations needed for a specific NN training are possible in the present mode.

For example, if a large NN must be trained with a training algorithm that requires a matrix inversion, just before the NN instantiation, the `Train-Mode` component will be notified to check if the matrix inversion is possible for the given NN size and NN training algorithm in the current mode. If the response is negative, then the user will be notified to upgrade his/her product with the **Network** feature, in order to perform this training. If the response is positive, the NN will be instantiated and trained. During the training, the `Train-Mode` component will be called upon for the matrix inversion, which will propagate the call to the component that had been selected during the NN configuration, i.e. either the `PC` or `Network` component.

4.5 Feature Interaction

After the first two transformation phases of the FArM method, there should exist ideally only functional features in the transformed FM. Furthermore, there should exist a balance between problem and solution domain related features. For each one of the features of the transformed FM, one architectural component must have been derived, who's specification reflects the specification of the feature it is implementing. Traceability links connect the features of the initial FM with those of the transformed FM.

All these factors provide a strong feature architecture mapping, but there exists one more factor that can further increase this mapping. Namely, the *feature interaction* (sect. 4.5.1). The communication needs between the features of the transformed FM can also be mapped to the respective architectural components in the form of component interfaces. This is performed in FArM through a series of discrete steps:

- Identification of Feature Interaction
- Optimization of Feature Interaction
- Interface Derivation

The first step of this transformation phase focuses on the identification of the interactions that exist between the features of the transformed FM. These interacts relations are indicators of the main communication needs between the features and respectively, between the architectural components. In the next step, the feature interactions are optimized. This is a vital precondition also for the optimization of the architectural component communication. This step contributes to the enhancement of the system maintainability and variability through e.g. decoupling and encapsulation of tightly coupled features with a high change probability. Finally, the interacts relations between the features are used to derive interfaces for the architectural components. Each of the steps of the third FArM transformation phase is described in more detail in the following sections.

4.5.1 Identification

Feature interaction is identified in FArM through the utilization of two constructs that are naturally present in a FM. Namely, *interacts relations* and *hierarchy relations*. Nonetheless, neither interacts relations, nor hierarchy relations can be used for the identification of feature interaction in their present form.

Interacts relations have been seen in existing works as an indirect influence between features (sect. 2.7). The concept of interacts relations in FArM is generalized to include any direct and indirect influence that a feature may have on another feature in the FM. Based on this generalization, three types of interacts relations can be identified as shown in table 4.5.

Interacts Relation Types	
Type	Description
Uses	A feature requires the functionality of another feature in order to operate properly and this functionality is readily provided by the other feature
Extends	A feature requires functionality from another feature, which is not readily provided by that feature
Runtime Excludes	Leads to the runtime exclusion of one of the features that participate in the relation

Table 4.5: FArM Types of Interacts Relations

These three types of interacts relations cover the entire probability space from the point of view of the functionality that a features may require or provide. Uses interacts relations cover all cases where the functionality needed is also *readily* provided by another feature. Extends interacts relations cover all cases where *more* functionality is required, leading to the extension of a feature and respectively to a change of the feature's behavior. Runtime excludes interacts relations cover all cases where *less* functionality is required, leading to the exclusion of functionality provided by another feature at runtime and thus also changing the feature's behavior.

The last two types of interacts relations can be and are transformed to uses interacts relations in FArM (see upcoming sections for details). This is possible, because in the case of an extends interacts relation, functionality is added to a feature, which is then simply *used* by the feature that required it. In the case of the runtime excludes interacts relations, a new feature can be added to decide which of the interacting features should be excluded at runtime. This feature *uses* the functionality of the interacting features and therefore resolves the runtime excludes interaction.

The transformation of the types of interacts relations to uses interacts relations allows the direct derivation of component interfaces. For instance, a feature using the functionality of another enables the addition of an interface to the component that implements that feature. Therefore, the feature interaction can be directly mapped to the architecture and thus enhance the feature-architecture mapping.

Apart from interacts relations, FArM utilizes *hierarchy relations* for the derivation of component interfaces. In most of the state of the art domain analysis methods, hierarchy relations are used to indicate a strong logical connection between the features participating in the relation. This logical connection is in most cases of pure structural nature. Nonetheless, a more strict definition of hierarchy relations can also allow the utilization of hierarchy relations for the derivation of component interfaces. In FArM, a hierarchy relation is used again as a structural element, but this time complying to exactly one of the types shown in table 4.6.

Hierarchy Relation Types	
Type	Description
Aggregation	The sub-feature is a part of its super-feature
Specialization	The sub-feature is a more concrete instance of its super-feature

Table 4.6: FArM Types of Hierarchy Relations

These two constraints placed upon hierarchy relations have a direct association to the architectural perspective of a software system, e.g. to object-oriented development principles. Under these circumstances, hierarchy relations can be also used, like interacts relations, for the direct derivation of component interfaces. For instance, if a feature is composed of a number of sub-features, then the respective super-feature architectural component can operate as a facade for the sub-feature derived components, i.e. the component uses the functionality of the other components to fulfill its facade responsibility. In the case of specialization, the component implementing the super-feature may also operate as a switch mechanism or it may hold the common functionality of the sub-feature components. Thus, the FArM hierarchy relations can be directly translated to interfaces for the component derived from a feature hierarchy.

This transformation step is performed with the following process: Initially, the extends and runtime excludes interacts relations are identified and transformed to uses interacts relations. Afterwards, based on the FArM definition of hierarchy relations, all pre-existing hierarchy relations of the FM are examined for validity. The hierarchy relations not complying to the FArM definition are effectively "broken" and uses interacts relations are placed between the separated features and their former super-feature. This process leaves only valid hierarchy relations and uses interacts relations in the FM.

After this transformation step, all feature interaction has been identified and captured in the form of interacts and hierarchy relations. In the next transformation step, the feature interaction is optimized with focus on maintainability and variability. The following sections present the aforementioned process of this transformation step in more detail.

Uses Interacts Relations

Uses interacts relations exist between features, when one feature requires the functionality of another feature in order to operate properly and this functionality is readily provided by the feature. An example of such an interacts relation can be found between the features related to the training of a NN in the NN-Trainer case study.

During the supervised training of a NN, a training pattern must be presented to the input of the NN and the NN must be activated to provide an output for this pattern. This output is compared to the expected value and based on a performance function, the deviation from this value is calculated. The calculated deviation is then fed into a training algorithm that adjusts the weights and biases of the NN, so as to minimize the error.

The PL developers examine this training scenario to identify interacts relations. Uses interacts relations exist between features that generically provide functionality needed by other features. Figure 4.15 shows the uses interacts relations identified in the NN-Trainer case study, based on the above scenario. Note that the figure shows a compact view of the FM, including only the features directly involved in the above scenario, along with their super-features. Extra details, e.g. cardinalities, other sub-features, etc., are excluded from the figure.

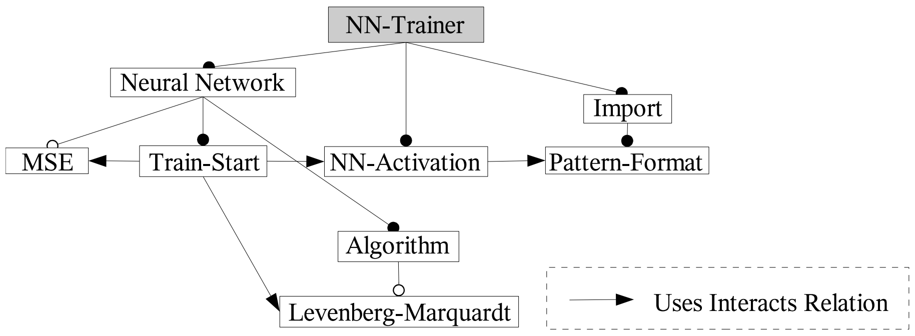


Figure 4.15: NN-Trainer uses interacts relations

As shown in fig. 4.15, the **Train-Start** feature coordinates the training of a NN. It triggers the NN activation by signaling the **NN-Activation** feature. The latter retrieves the current training pattern from the **Pattern-Format** feature and calculates the NN output. The result is returned to the **Train-Start** feature, which now triggers the **MSE** feature. MSE stands for Mean Square Error and the **MSE** feature is the one that implements the performance function and calculates the deviation from the expected value. Finally, the **Levenberg-Marquardt** feature is responsible for calculating and updating the weights and biases of the NN.

Each of the aforementioned features generically provides the functionality, which is demanded from it. This can be found in their specification. For instance, the **Pattern-Format** feature is responsible for importing files and converting them into input patterns. It is also responsible for providing these patterns to any other feature that needs them. The same holds for the other features presented in this scenario. If a feature requires this readily available functionality, then the features are said to have a *uses interacts relation.*

In the given example, there existed only unidirectional uses interacts relations. Nevertheless, uses interacts relations may also be bidirectional, e.g. in the cases where two features require functionality from each other in order to operate properly.

As already mentioned, this is the main kind of interaction that may eventually be present in a FArM FM. This type of interaction indicates the actual communication needs between two features and can be directly used for the derivation of component interfaces. The interacts relations presented in the following section are transformed to uses interacts relations.

Extends Interacts Relations

Extends interacts relations exist, when a feature requires functionality from another feature, which is not readily provided by that feature, consequently leading to the extension of the feature. In other words the feature's behavior is change in some way due to the presence of the other feature. This kind of interacts relation is shown in the context of an example from the NN-Trainer case study.

During the second transformation phase in section 4.4, the specification of the **License** feature was enhanced to satisfy the *External License Manager* architectural requirement. This enhancement raised the need for each feature of the PL to be license-protected through the inclusion of special method-calls, as shown in listing 4.1. Thus, the **License** feature indirectly affects all other features of the PL. It enforces that each feature interacts with it in order to operate. This can be expressed graphically in FArM as shown in fig. 4.16. The asterisk of fig. 4.16 indicates that each other feature of the FM (not explicitly shown in the figure) has an extends interacts relation to the **License** feature. The **+** sign above the arrow is used to indicate the extension made to the uses interacts relation of fig. 4.15.

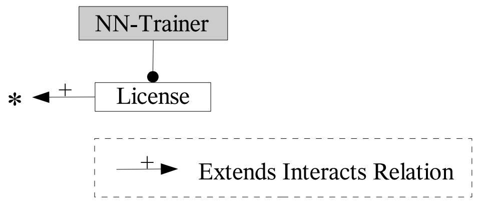


Figure 4.16: NN-Trainer extends interacts relations

For example, the **Levenberg-Marquardt** algorithm feature must interact with the **License** feature in order to operate. This implies that the **Levenberg-Marquardt** feature must be *extended*, so as to provide the functionality needed by the **License** feature. This kind of interacts relation is said to be an *extends interacts relation*. Like in the case of uses interacts relations, extends interacts relations can be either unidirectional or bidirectional.

The extends type of interacts relation is transformed to a uses interacts relation in FArM. This is the simple process of extending each feature's specification with the specification of the interacts relation and choosing the right direction for the interaction based on the features' needs.

For the example of the **License** feature, all feature specifications that have an extends interacts relation are extended to indicate that they must use the **License** feature functionality for the enforcement of the licensing policy. Thus, the extends interacts relations are transformed to uses interacts relations pointing to the **License** feature, as shown in fig. 4.17. The direction of the interacts relation is set by the fact that the **License** feature needs to have no knowledge of which features make use of its functionality, while all other features do need to known the provider of the functionality.

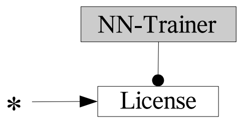


Figure 4.17: Transformation of the extends interacts relations

Runtime Excludes Interacts Relations

A runtime excludes interacts relation is an interacts relation that leads to the runtime exclusion of one of the features that take part in the relation. As in the case of extends interacts relations, again the presence of another feature practically influences the behavior of the feature. An example of such an interaction can be found between the **MSE** and **MAE** features of the NN-Trainer FM.

As shown in the example for the uses interacts relations, the **MSE** feature implements the algorithm that defines the performance of a NN during its training. The **MAE** feature stands for Mean Average Error and refers to another performance function that may be selected for the training of a NN.

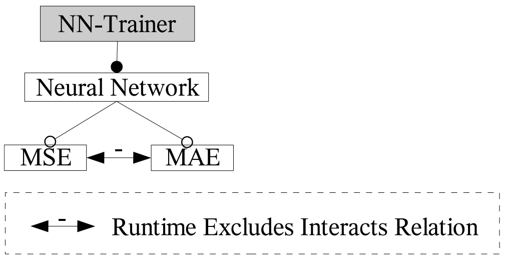


Figure 4.18: NN-Trainer runtime excludes interacts relation

When the **Train-Start** feature in fig. 4.15 needs to trigger the calculation of the

NN training performance, it must ”know”, which feature is responsible for this calculation. Since both the **MSE** and **MAE** features provide this functionality and they may both be present in the system, it must be distinguished at runtime, which of the features is to be activated.

This case illustrates a runtime excludes interacts relation. If it is not clear, which of the **MSE** and **MAE** features is to be activated, then the behavior of the system is unpredictable. Therefore, only one of the **MSE** and **MAE** features can be active at runtime. In other words, the two features are mutually exclusive at runtime. This kind of interacts relation is shown graphically in fig. 4.18. The – sign in the relation indicates the runtime mutual exclusion of the two features.

In contrast to the two previous types of interaction, runtime excludes interacts relations are always bidirectional, i.e. both features that take part in such a relation influence the other feature.

Similarly to the extends interacts relations, the runtime excludes interacts relations are transformed to uses interacts relations. For this purpose, a new feature is created to decide at runtime, which of the alternatives is to be activated at runtime. The runtime excludes interacts relation is then replaced by a uses interacts relation pointing from the new feature to the alternatives.

In the case of the **MSE** and **MAE** features, a new **Performance** feature is added with the responsibility to activate the proper feature at runtime (fig. 4.19). This feature must be configured by the user before the training of the NN with the proper performance function.

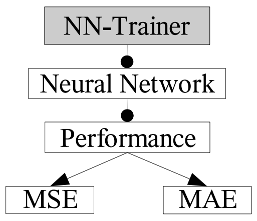


Figure 4.19: Transformation of the runtime excludes interacts relation

Finding Interacts Relations

The PL developers are guided by the FArM interacts relation types to find the interactions between the features in the FM. This process is rather domain specific and therefore cannot be entirely methodized. Nonetheless, a number of guidelines can be provided that further support the finding of interacts relations.

One obvious approach is to utilize domain knowledge. The feature analysts can identify feature interactions by making use of their experience in the domain. Another approach is the utilization of the *requires* relations. Requires relations are very frequently created during the domain analysis process and are generically supported by numerous domain analysis methods, e.g. FODA [KCH$^+$90]. The requires relations are in most cases indicators of uses interacts relations. Feature multiplicities can also serve as indicators of feature interactions. Features having a multiplicity of `1..*` and are part of a FArM aggregation hierarchy relation, are likely to have uses interacts relations with each other. Features having a multiplicity of `1..*` and are part of a specialization hierarchy relation, must have FArM runtime excludes interacts relations with each other. The former case is a natural consequence of an aggregation hierarchy relation. Features taking part in such a relation most often cooperate to accomplish a complex task. The latter case is a definite indicator for a runtime excludes interacts relations, since the features perform similar tasks in different ways. An example of such a case was seen in the **MSE** and **MAE** feature hierarchy.

Additionally, the feature analysts can take advantage of the system use cases. Examining use case scenarios may bring numerous interactions to the surface, belonging to any of the three FArM types. An example of the identification of uses interacts relations was given for the NN training use case scenario.

The architecture team may also use its architectural expertise to identify interactions between components in the architectural level. For example, the `Network` component introduced during the resolution of the *network training* architectural requirement (sect. 4.4) could lead to the identification of interactions between the **Network** feature and other features implementing memory intensive algorithms.

The whole searching process can be further supported through simple searching tools. For example, the PL developers may utilize a text searching tool that will run through the feature specifications and search for similar lexical structures, e.g. nouns and adjectives or verbs and adverbs. Similar lexical constructs between different feature specifications indicate possible interactions, which can then be more closely examined for classification into the FArM interaction types.

Hierarchy Validation

After finding feature interactions based on the types of interacts relations, the PL developers concentrate on the identification of feature interactions on the basis of hierarchy relations. In order to accomplish this task, all pre-existing hierarchy relations are examined for validity based on the FArM definition for hierarchy relations. Namely, a hierarchy relation must have a structural role, but only in the form of either an *aggregation* or a *specialization*. Additionally, in the case where a feature has more than one hierarchy relation, the type of the hierarchy relations must remain consistent throughout. For example, if a feature has an aggregation relation with one of its sub-features, then it must not have a specialization relation to any of its sub-features.

This extra restriction placed upon hierarchy relations focuses on the enhancement of the system's maintainability and the preservation of the system's conceptual integrity. On the one hand, the component derived from the super-feature of an invalid hierarchy relation has a number of different responsibilities, depending on the specification of its sub-features. The fulfilment of these responsibilities increases the complexity of the derived components, thus reducing their maintainability. On the other hand, the inconsistencies that arise have a negative impact on the system's conceptual integrity.

Invalid hierarchy relations are broken and the sub-features of the relation are placed under the root feature of the FM. Additionally, a uses interacts relation is created between the single features and their former super-feature. The semantics of the interacts relation should reflect the semantics of the broken hierarchy relation. The direction of the interacts relation depends from the case at hand. This is done due to the fact that the broken hierarchy relation indicates some kind of domain-specific relation between the features, which must be taken into consideration during the development process. The interacts relation created during the breaking of an invalid hierarchy relation plays an important role during the next step of this transformation phase, when the optimization of the hierarchy relations takes place (sect. 4.5.2). During that transformation step, the interacts relation can be used to properly order the single feature under its former super-feature.

In the NN-Trainer case study, the initial FM has been developed based on the FODA domain analysis method. The latter has the notion of hierarchy relations similar to the aggregation/specialization concepts of FArM [PS94]. An example of an aggregation hierarchy relation from the NN-Trainer case study is shown in fig. 4.20. This example shows that the **Neural Network** feature is composed of a number of elements, e.g. a **Neuron Distance**, a **Topology**, etc.

An example of a specialization hierarchy relation is given in fig. 4.21. The **Import** feature refers to the ability of the NN-Trainer to import patterns and NNs of specific format. Both **Pattern-Format** and **NN-Format** are special types of importing,

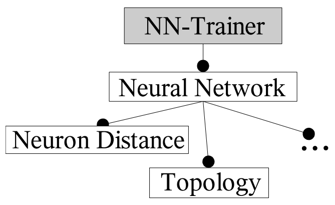


Figure 4.20: An aggregation hierarchy relation

i.e. pattern-importing and NN-importing respectively.

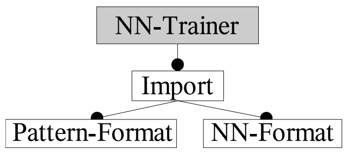


Figure 4.21: A specialization hierarchy relation

An example of an invalid hierarchy relation is found under the **Pattern-Format** feature of the **Import** feature hierarchy. Fig. 4.22 illustrates the **Pattern-Format** hierarchy in more detail. This feature hierarchy does not comply to the FArM definition of a hierarchy relation, given at the beginning of this section. Namely, the hierarchy relations of the super-feature with its sub-features are not consistent. The **Binary** and **Text** features represent specialized pattern formats, whereas the **Pattern-Format** feature has an aggregation relation to the **Structure** feature. The latter indicates solely that the pattern format can be structured with spaces, newlines, etc. Thus, the **Pattern-Format** feature has both an aggregation and specialization relation with its sub-features.

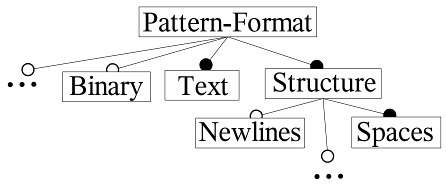


Figure 4.22: An invalid hierarchy relation

The hierarchy relation is broken according to the aforementioned FArM rule, placing the **Binary**, **Text**, **...** features as single features under the **NN-Trainer** root feature. Additionally, a uses interacts relation is created between the features and the **Pattern-Format** feature (fig. 4.23). In this case, the direction of the interacts relation targets the single features and eventually results in the `Pattern-Format` component effectively using e.g. the `Text` component for importing a textual training pattern.

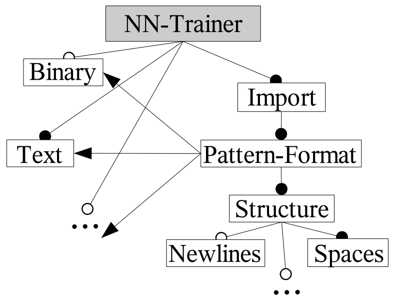


Figure 4.23: A broken invalid hierarchy relation

4.5.2 Optimization

At this step of the third FArM transformation phase, ideally, all feature interactions have been captured in the form of valid hierarchy and uses interacts relations between the features of the FM. The goal of this transformation step is to optimize the feature interaction. This is done through a number of processes attempting to achieve the right balance between uses interacts relations and hierarchy relations.

Although both kinds of relations are indicators of feature interaction in FArM, they do possess different properties. Hierarchy relations have, additionally to interacts relations, also a structural role. This structural role of hierarchy relations is translated on the architectural level as increased encapsulation and decoupling, thus enhancing the system maintainability and variability. A few examples are the use of super-features for the derivation of components that act as containers of common sub-feature functionality, for the encapsulation of the components implementing its sub-features or as components that act as switches between them.

Interacts relations on the other hand, may reach across the hierarchy structure and directly connect two features. This direct connection is translated on the architectural level as a direct communication between the respective components, which in some cases may increase the performance of the system in time or resource-critical situations. Nevertheless, this performance increase is most frequently accompanied by a maintainability penalty.

During the identification of feature interactions the main activity was the creation of uses interacts relations and the breaking of invalid hierarchy relations. These processes lead normally to an asymmetry between the number of these two kinds of relations. Thus, this transformation phase strives to effectively restore a balance between them by increasing the number of hierarchy relations and at the same time decreasing the number of uses interacts relations.

This is done through the following processes:

- Hierarchy Relation Derivation
- Hierarchy Relation Enhancement
- Normalization of Interacts Relations

The first process of *Hierarchy Relation Derivation* is concerned with the utilization of the existing uses interacts relations for the derivation of hierarchy relations. This process directly reduces the number of uses interacts relations, while increasing the number of hierarchy relations. The second process of *Hierarchy Relation Enhancement* is concerned with the merging of the remaining uses interacts relations with existing hierarchy relations. This process reduces the number of uses interacts relations, while maintaining a stable number of hierarchy relations. Finally, in a last effort to further reduce the number of uses interacts relations, the remaining features are examined for high interaction with other features based on their number of interacts relations and are respectively transformed. Each of these processes are described in detail in the following sections.

Hierarchy Relation Derivation

The first process of this transformation step is the derivation of hierarchy relations based on interacts relations. This transformation process provides an immediate transformation of interacts relations into hierarchy relations. This fact allows eventually the derivation of components that can enhance the maintainability of the system by e.g. encapsulating the components derived from their sub-features. Additionally, the reduction of interacts relations leads also to the minimization of component communication across the software system. The structural cohesiveness of the system is thus increased.

The derivation of hierarchy relations is performed through the examination of all existing interacts relations. The PL developers attempt to identify interacting features, which also comply to the FArM hierarchy types. Namely, aggregation or specialization. If there do exist features that have both an interacts relation and also a logical aggregation or specialization relation, then the interacts relation is a good candidate to be transformed into a hierarchy relation.

The probability of interacting features having also a logical aggregation or specialization relation at this point in FArM is not minimal. On the contrary, there may exist numerous cases where this phenomenon exists. This is due to the fact that FArM does not make any assumptions of the domain analysis method used for the development of the initial FM. Because of this, the identification of interacts relations did not take into consideration the hierarchy structure of the FM. Thus, there may exist several features whose interacts relations can be directly transformed into a hierarchy relation.

The transformation of an interacts relation to a hierarchy relation is done based on the semantics of the logical relation between the features, the direction of the interacts relation and possible pre-existing hierarchy relations that each of the features may have. Based on the semantics of the logical relation between the two features, the one specializing or being aggregated from the other should be placed as a sub-feature. Based on the direction of the uses interacts relation, the feature being used should be placed as a sub-feature under the feature using it. In the case of bidirectional interacts relations, the PL developers must base their decision on the specialization or generalization relation alone. Finally, the validity of the pre-existing hierarchy structure for each of the features must be preserved. For example, a feature already having a specialization relation may only receive a new sub-feature if it can also have a specialization relation with that feature. If this is not directly possible, new features must be added to preserve the validity of the pre-existing hierarchy structure.

Examples of the aforementioned cases can be found in the NN-Trainer case study. Fig. 4.24 illustrates the derivation of a specialization relation. At the left part of the figure, the **Template** feature under the **Neural Network** feature refers to the ability of the system to provide templates for the design of a NN. The importance of this feature is also evident from the fact that it is a mandatory feature. Namely, the **Template** feature significantly reduces the effort of setting the right parameters for a NN for a variety of NN architectures. In order for the feature to accomplish its task, it has two uses interacts relations with the **Feed-Forward** and **Self-Organizing** features respectively. The latter can set the right parameters for the instantiation of NNs, complying to the feed-forward backpropagation and self-organizing maps NN architectures.

In this transformation process, the PL developers identify that there exists a specialization relation between the three features. More precisely, the **Feed-Forward** and **Self-Organizing** features are both special cases of the **Template** feature. Ad-

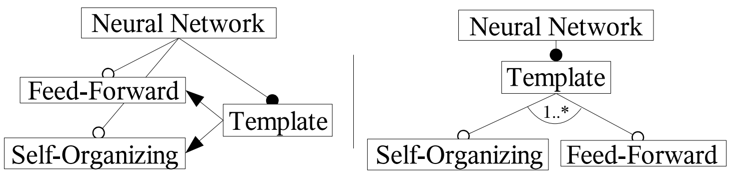


Figure 4.24: Derivation of a specialization hierarchy relation

ditionally to the nature of the specialization relation, the direction of the interacts relations indicates that the features are to be placed as sub-features of the **Template** feature. Furthermore, the validity of the pre-existing hierarchy relations of all three features is not broken. Thus, the interacts relations between the features are transformed into hierarchy relations, as shown in the right part of fig. 4.24. The **Feed-Forward** and **Self-Organizing** features become sub-features of the **Template** feature. The cardinality of `1..*` is also added for domain specific reasons, i.e. a customer may select during the configuration of a product either the **Feed-Forward** or **Self-Organizing** features or both.

An example for the derivation of an aggregation hierarchy relation can be found during the examination of the interacts relations of fig. 4.23. These interacts relations resulted from the breaking of the invalid hierarchy relations between the **Pattern-Format** feature and the **Binary**, **Text**, **...** features respectively. The PL developers identify in this case that there does exist a logical relation between these features, i.e. a specialization relation. For example, a pattern can be of binary format. Therefore, the interacts relations should be transformed to specialization hierarchy relations. Additionally, based on the direction of the interacts relations, the **Binary**, **Text** and **...** features should be placed as sub-features under the **Pattern-Format** feature. Nonetheless, this transformation cannot be directly performed, since this would break the validity of the pre-existing hierarchy relation of the **Pattern-Format** feature with the **Structure** feature.

As already mentioned, in such cases FArM dictates the addition of a new feature to enable the creation of the hierarchy relation. In this example, the **Pattern-Format** feature is transformed to the more general **Pattern** feature and a new **Format** feature is added under this feature, as shown in fig. 4.25. Now, the **Binary**, **Text** and **...** features can be placed as sub-features under the new **Format** feature. The interacts relations are transformed to hierarchy relations and the validity of the hierarchy structure is preserved, i.e. the general **Pattern** feature has now only aggregation hierarchy relations to its **Format** and **Structure** sub-features. For a more formal specification of the **Pattern** feature refer to Appendix B.

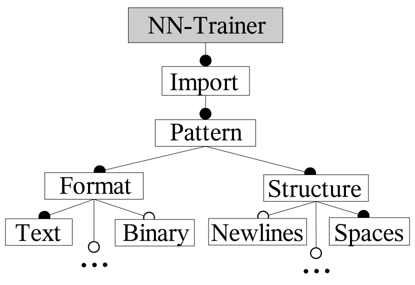


Figure 4.25: Derivation of an aggregation hierarchy relation

Hierarchy Relation Enhancement

During the derivation of hierarchy relations, interacts relations were directly transformed into hierarchy relations. This process led to an immediate reduction of the number of interacts relations and to an increase in the number of hierarchy relations. This fact localized feature interactions to small hierarchy structures, while minimizing interactions across the system. The Hierarchy Relation Enhancement transformation process focuses on the merging of interacts relations with existing hierarchy relations. The main goal of this process is to further reduce the number of interacts relations and thus increase system maintainability.

FArM defines an *interacts relation delegation process* for the enhancement of hierarchy relations. During this process, interacts relations are delegated to features higher on the FM hierarchy and their hierarchy relations are respectively enhanced. Such a hierarchy enhancement is shown in fig. 4.26. In the left part of the figure, the sub-feature (lower feature) has an interacts relation to another feature. The latter can be any other feature of the FM, e.g. even a neighbor sub-feature. On the right part of the figure, this interacts relation is delegated to the super-feature (upper feature) and the hierarchy relation between the features is enhanced.

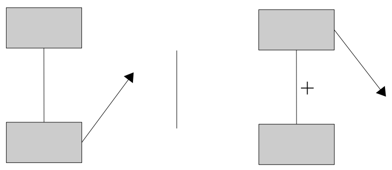


Figure 4.26: Hierarchy enhancement

This enhancement enables the PL developers to decide which percentage of the feature interaction specification is to be performed by the super-feature and which percentage should be delegated back to the sub-feature. For example, if the super-feature is to act as a facade or as a switch, then most of the interacts relation specification is merged with the hierarchy relation. If the super-feature is meant to further encapsulate the sub-feature by performing e.g. some kind of information pre-processing, then it receives a larger part of the feature interaction and the rest is used to enhance its hierarchy relation to the sub-feature.

The interacts relation delegation process begins by the ”lowest” features of the FM and works its way up the FM hierarchy towards the root feature. Now the question arises, when the delegation process should cease. The answer to this question is: When there are no more additional benefits from the application of the delegation to higher levels.

Theoretically, if the delegation process is applied throughout the entire FM, then communication would be performed only through hierarchy relations. For example, if a feature on a lower level needs to communicate with another feature on another branch, then it should send a message to its super-feature, that to its super-feature, etc., until the message reaches the root-feature and then the hierarchy-chain down again to the desired feature (fig. 4.27).

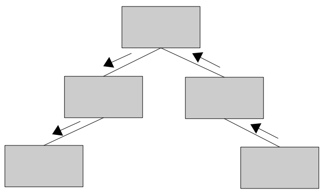


Figure 4.27: Feature interaction exclusively through hierarchy relations

This kind of communication has the advantage of high encapsulation, but on the cost of performance. It is evident that a balance must be found between performance and maintainability. Experience with FArM has shown that the optimal point to cease the application of the delegation process is the one where the super-feature acts merely as a facade or a switch for its sub-features. In this case, no further information encapsulation can be performed in the next higher layer. At this point, the continuance of the interacts relation delegation process ceases to provide any further maintainability advantages through encapsulation and decoupling.

These various patterns of communication that occur during the application of the interacts relation delegation process are possible, because of the nature of the FArM hierarchy relations. That is, the specialization and aggregation hierarchy relations between features enable the super-feature to take on some of the interactions of its sub-features. This facilitates the provision of a natural encapsulation and/or variability mechanism on the architectural level, through the derivation of components from the super and sub-features.

An example of the application of the interacts relation delegation process is shown in fig. 4.28. The figure shows a part of the interacts relations for the training of a NN (sect. 4.5.1). In the left part of the figure, the **Train-Start** feature interacts with the **Levenberg-Marquardt** feature for the activation of the training algorithm for a given NN. The application of the FArM interacts relation delegation process leads to the redirection of the interacts relation to the super-feature of the **Levenberg-Marquardt** feature, i.e. the **Algorithm** feature. Through this delegation, the **Algorithm** feature receives the extra responsibility to identify and activate the right training algorithm. On the architectural level, the `Algorithm` component will provide a switch mechanism between the various algorithms. The same component could also e.g. perform some kind of data pre-processing before activating certain algorithms. This fact allows the easier accommodation of new training algorithms and even the dynamic switching between them, thus increasing the components' maintainability and variability.

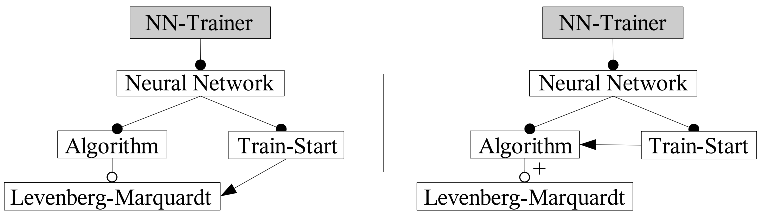


Figure 4.28: Interacts relation delegation

Note also that although it is possible, there is no point into further applying the interaction delegation process in the above example. Yet another interaction delegation would lead to the creation of an interacts relation between the **Neural Network** and **Algorithm** features. In such a case, the interacts relation would be entirely merged with the hierarchy relation, as shown in fig. 4.29. This would now imply that the **Neural Network** feature is responsible for notifying the **Algorithm** feature to activate the proper training algorithm. Such a communication path for the activation of the proper training algorithm does not further encapsulate the training algorithms, rather just adds additional overhead in the activation process. Thus, the level of the **Algorithm** feature is the proper one for ceasing the interaction delegation process.

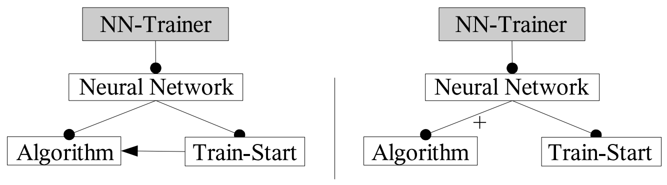


Figure 4.29: Improper continuance of the interaction delegation process

Normalization of Interacts Relations

At this stage of the feature interaction phase, several of the interacts relations have been either transformed to or have been merged with hierarchy relations. In a last attempt to further reduce the number of interacts relations, FArM performs a *per feature normalization of interacts relations.*

In this transformation process, features having a large number of interacts relations are examined for transformation. A large number of interacts relations indicates that a feature has a lot of dependencies on the other features of the FM. Respectively, the architectural component derived from this feature will also have the same dependencies from the other components in the system architecture. If the feature has a high change probability in the future, then it is likely that these changes propagate to the dependant architectural components in the software architecture. This fact has a negative influence on the system's maintainability.

In such cases, the FArM *create* elementary transformation is applied. With this elementary transformation new features are created to take on the responsibilities of the original feature. Through this transformation, the overall number of interacts relations remains constant, while the per feature number of interacts relations is normalized. This has in turn a direct influence on the maintainability of the system, since the new features have, on the one hand, a smaller change probability and on the other hand, their implementation complexity is reduced, compared to the original feature.

The identification of a high interacting feature depends on the average number of interacts relations of all features in the FM. At first, the PL developers calculate the number of interacts relations for each feature in the FM. These numbers are then summed and divided with the number of features in the FM. The features having a larger number of interacts relations respective to the average are then considered for transformation.

In FArM, the calculation of the number of interacts relations for a feature is done through the addition of the uses interacts relations that a feature has, plus the number of interacts relations of its direct sub-features. For example, the super-feature in fig. 4.30 has a total of 6 interacts relations. This number is the result of the sum of the 2 interacts relations of the feature itself, plus the 2 interacts relations of each of its direct sub-features.

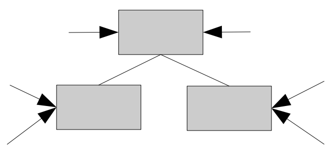


Figure 4.30: Calculation of interacts relations

This way of performing the calculation of a feature's interacts relations is based on the strong mapping between features and architectural components. A super-feature is used in FArM to derive an architectural component that can directly communicate with the components derived from its sub-features. Therefore, the interacts relations of the sub-features have a direct influence on the super-feature. For instance, the super-feature may be responsible for the common functionality of its sub-features and thus provides services to them, when they directly interact with other features. This way, the interacts relations of the sub-features indirectly influence the super-feature. Therefore, sub-feature interacts relations must be taken into consideration for the calculation of the number of interacts relations of their super-features.

The final decision to transform a feature depends on the number of interacts relations and the change probability of the feature. The decision for a feature transformation is left to the judgement of the PL developers and must be made individually for every feature. A summary of the general rules that apply are shown in table 4.7.

Transformation Decision Rules	
Rule	Description
High Interaction	The larger the deviation from the average, the higher the transformation probability
Change Probability	The higher the probability for future change, the higher the transformation probability

Table 4.7: Decision rules for the normalization of interacts relations

Nonetheless, it may not always be possible to apply the create elementary transformation to every feature of the FM that has a high deviation from the average number of interacts relations. This may be due to the inability of creating new features to take on the responsibilities of the original feature. It may also be the case that the newly created features need more interacts relations as the original feature. This would then increase the overall number of interacts relations.

An example of the normalization of interacts relation can be found in the NN-Trainer case study. The **Neural-Network** feature has a large number of direct sub-features as shown in fig. 4.31. Although the number of interacts relations for each of these features does not exceed the average, the **Neural Network** feature has a high deviation from the average number of interacts relations. This is due to the way interacts relations are calculated in FArM, namely, as the sum of the interacts relations of the direct sub-features.

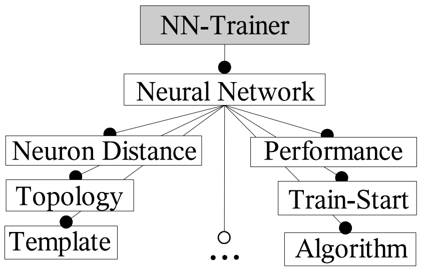


Figure 4.31: The **Neural Network** feature hierarchy

Therefore, the **Neural Network** feature becomes a candidate for transformation. The PL developers evaluate in a consecutive step the feature's change probability. The **Neural Network** feature plays a main role in the NN-Trainer PL. It practically reflects the ability to design and train a NN. This can also be seen from its sub-features. For example, the **Neuron Distance**, **Topology** and **Template** features are all related to the design of a NN, while the **Performance**, **Train-Start** and **Algorithm** features are related to the training of a NN. Thus, the **Neural Network** feature reflects the central functionality of the NN-Trainer PL. This is the functionality that will need to be extended or optimized in future versions of the PL.

Because the **Neural Network** feature has both a large number of interacts relations and a high change probability, it is transformed with the create elementary transformation. The PL developers identify the double role that the **Neural Network** feature plays and create the **NN-Design** and **NN-Train** features to replace it. The design-related sub-features of the **Neural Network** feature are placed under the **NN-Design** feature and the train-related sub-features under the **NN-Train** feature as shown in fig. 4.32.

This transformation allows for a reduction of the per feature interacts relations. The interacts relations of the newly created **NN-Train** feature allow for the en-

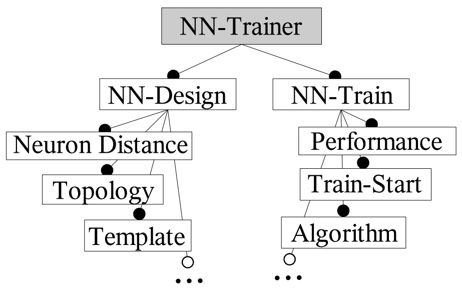


Figure 4.32: Normalization of the **Neural Network** feature's interacts relations

capsulation of the training functionality of the NN-Trainer PL. Accessing training functionality may for example take place through the **NN-Train** feature when no direct access to the sub-features of the **NN-Train** feature is required. Therefore, changes to the way a training is conducted can be hidden from the rest of the features and thus, more easily changed in the future. The same holds for the **NN-Design** feature hierarchy.

Another example of a feature with a large number of interacts relations is the **License** feature in fig. 4.13 (sect. 4.5.1). The **License** feature interacts with every feature of the FM in order to impose the desired licensing policy. Therefore, it has the largest deviation from the average number of interacts relations from all features in the FM. Nonetheless, for the **License** feature no meaningful features can be created to take on its responsibilities. The **License** feature is itself an encapsulation layer for the third party license manager. Further "division" of the encapsulation layer would consequently lead to no extra advantages. Therefore, no expedient components can result from the application of the create elementary transformation and no further reduction of the **License** feature interactions can be performed.

Nevertheless, there may exist certain cases where the performance of the software system is more important than its maintainability. A few example domains are hard real-time systems, computer games, etc. In such domains, the large number of interacts relations can be seen as a chance for performance optimization. In this case, a large number of interacts relations indicates that a system component needs to access or be accessed by a large number of other components. Most frequently, access calls are performed through the component's interface, they are processed and are then propagated to the sub-components in its internals. A performance optimization could be achieved if the access calls are sent directly to the sub-component.

This kind of performance optimization can be achieved with the FArM *merge* elementary transformation. With the merge elementary transformation, a feature with a high number of interacts relations is merged with the features using it or with the features it is using. This way, a direct access to its internals is achieved that gives a slight boost in performance.

An example can be found in the game development domain, where direct access to graphic card functionality is allowed. In these cases the developers renounce the use of a hardware abstraction layer, which would be respectively implemented into a FArM feature, in order to achieve the highest performance.

Handling Pre-Existing Hierarchy and Interacts Relations

An issue that arises during the FArM transformation phases is the handling of pre-existing hierarchy and interacts relations during the FArM transformation phases. During the transformation phases, features may be directly resolved, merged with others, features may be reordered or new features may be created to replace other features. The features influenced by these transformations may have pre-existing hierarchy or interacts relations. The question that now arises is: ”How are the hierarchy and interacts relations of these features handled?”

Hierarchy relations may stem from the domain analysis method. These hierarchy relations may be influenced even at the beginning of the FArM iterations. For example, a feature that has a hierarchy relation identified during the domain analysis method may be directly resolved. For this kind of hierarchy relations FArM defines no direct measures. The hierarchy relations that originate from the domain analysis are simply broken without any further processing. This is due to the fact that FArM explicitly focuses on hierarchy relations in the third transformation phase, where feature interactions are identified and optimized. In this transformation phase, valid FArM hierarchy relations are created based on a systematical process. This process eliminates the need for preserving domain related hierarchy relations.

Nonetheless, FArM hierarchy and interacts relations may be influenced throughout the FArM transformation phases. In such cases a number of rules are identified for their handling. These rules are categorized based on the various transformations that may occur.

During the direct resolution of a feature, its hierarchy relation is documented and added to the definition of the traceability link between the feature and the root feature of the transformed FM. In the left-hand-side FM of fig. 4.33, the lower feature is directly resolved, leading to the breaking of its hierarchy relation. The latter is captured by the traceability link between the feature and the root feature of the transformed FM, which is indicated by the plus sign over the traceability link.

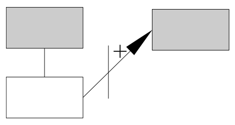


Figure 4.33: Handling of FArM hierarchy relations during a direct resolution

During direct resolution, interacts relations are redirected to the super-feature. Fig. 4.34 shows the various scenarios. In the FM on the left part of the figure, the lower feature has two interacts relations. It is being used by another feature of the FM (left interacts relation) and it uses another feature (right interacts relation). After its direct resolution, these interacts relations are respectively redirected to its super-feature. In the right part of the figure, the former super-feature has now received both interacts relations and respectively the responsibilities of its former sub-feature.

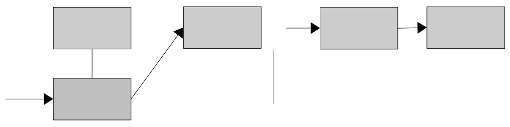


Figure 4.34: Handling of FArM interacts relations during a direct resolution

In the case of merging, pre-existing hierarchy relations are transformed to interacts relations as shown in fig. 4.35. The feature in the transformed FM (right part of figure) with the plus sign, is the result of the merging of the sub-feature and single feature shown at the left part of the figure. The hierarchy relation between the sub-feature and its super-feature is now transformed into an interacts relation between the former super-feature and the feature resulting from the merging transformation. Interacts relations in the case of merging are merely "inherited" by the feature that results from the merging. This is a natural consequence, since the resulting feature is extended and is thus capable of handling any interactions the merged features might have had with others.

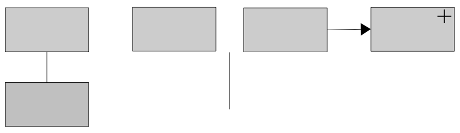


Figure 4.35: Handling of FArM hierarchy relations during merging

The case of the creation of features for the replacement of another can be discussed in the example of the resolution of the **Neural Network** feature, given in the previous section on the normalization of interacts relations. In this example, the **Neural Network** feature was replaced by two other features, the **NN-Design** and **NN-Train** features. In such cases, the hierarchy relations that a feature may have had are divided between the new features as shown in fig. 4.32. This process is largely dependent on the case at hand and relates to the responsibilities of the newly created features. Similarly, interacts relations formerly pointing to the transformed feature must now be distributed among the newly created features.

Finally, in the case of reordering, hierarchy relations may be broken or new hierarchy relations may be created. The former case may occur because of an invalid hierarchy relation, while the latter during the creation of a new valid hierarchy relation. Examples of these cases have been given in figures 4.23 and 4.25 respectively.

In these examples, the **Binary**, **Text**, **...** sub-features were found to have an invalid hierarchy relation to the **Pattern-Format** feature. This led to the breaking of these hierarchy relations. In such cases, the former hierarchy relations are transformed to interacts relations as shown also in fig. 4.23. Additionally, if one of the features has itself sub-features, then these "follow" practically the feature, i.e. they neither become single features, nor are they bound with a hierarchy relation to the super-feature of the broken hierarchy. The interacts relations of the single features remain unchanged.

During the creation of a new hierarchy relation, the hierarchy relations are transformed dictated by the transformation process as in the case of **Binary**, **Text**, **...**, features shown in fig. 4.25. In this case also, the interacts relations that the features may have remain unchanged.

4.5.3 Interface Derivation

In this last step of the feature interactions transformation phase, the PL developers concentrate on the derivation of component interfaces. This step closes the circle of the component derivation process based on the transformed FM. For each of the features of the transformed FM, exactly one architectural component is defined. Based on this mapping, the feature interactions can be directly used for the definition of the communication needs of the architectural components. This is done on the basis of the hierarchy and uses interacts relations among the features. These relations have been identified and optimized for maintainability and variability in the previous two steps of this transformation phase. Furthermore, compromises regarding these two factors have been made were necessary in favor of performance.

This transformation step is going to derive interfaces for the architectural components by means of the various interacts and hierarchy relations between features. The next sections explore the various possibilities and examples are given from the NN-Trainer case study.

Interacts relations

At this stage of FArM, the interacts relations that exist between the features of the FM are *uses* interacts relations. These interacts relations allow for the direct derivation of interfaces. This is because the uses type of interacts relation per definition dictates which functionality a feature provides and which functionality it requires. This information can be directly mapped to the respective provides and requires interfaces for the component that implements the feature. This fact increases the feature-architecture mapping between features and the architecture.

In this transformation step, the PL developers utilize the interacts relations to derive the component interfaces. This process cannot be performed directly, since it naturally requires input from the solution domain. The PL architects are involved in this process by providing their domain specific knowledge to complement the information taken from the interacts relations. In other words, the interacts relations define the interface context and rationale, while the PL architects fill in the details and specialize the interfaces based on solution domain knowledge.

At this point, the PL developers utilize the direction of the interacts relations for concluding if a certain interface should be a required or provides interface. This can be rather intuitively deduced from the direction of the arrow of the graphical form of FArM interacts relations. In the simple case of fig. 4.36, the left feature *requires* certain functionality, which is *provided* by feature on the right.

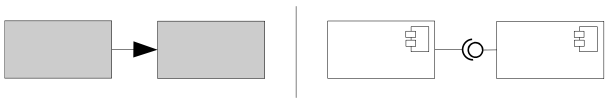


Figure 4.36: Derivation of requires or provides interfaces

Hierarchy Relations

The placement of a feature in the FM hierarchy can be just as well utilized for the direct derivation of component interfaces as uses interacts relations. The PL developers can now take advantage of the "component"-oriented nature of FArM hierarchy relations for the derivation of interfaces. Namely, they can exploit the fact that a feature can have either an aggregation or a specialization relation with its sub-features. This fact allows the PL developers to use the components derived from the super-features as facades for decoupling of functionality, for the encapsulation of sub-feature common functionality or as placeholders for functionality switching mechanisms. All these various roles that a FArM super-feature can play can be reflected as component interfaces in the software architecture of the system.

The various possibilities can be directly identified from the FM hierarchy structure. If a hierarchy relation is an aggregation interacts relation, then it is most likely for the component derived from the super-feature to play the role of a facade. This is because diverse functionality is provided by the sub-features, which is very frequently combined to accomplish a more complex task. Therefore, in the case of an aggregation hierarchy relation, the sub-feature components provide interfaces to the super-feature component, which reveal their functionality. The super-feature component must then provide a unified interface to the "outer world" that eases the use of the sub-feature functionality.

In the case of a specialization relation, the super-feature component can be most frequently used for the encapsulation of common functionality and/or as a switching mechanism. In the former case, the component derived from the super-feature implements an interface that is common to the sub-feature components. Within this interface, a number of operations are performed that are needed from each of the sub-feature components.

In order for a feature to be used as a switching mechanism, in addition to a specialization hierarchy relation, the feature must also fulfil the following preconditions:

- The feature has more than one sub-feature
- More than one sub-feature can be selected by the user, i.e. non-`1` cardinality

If all the above preconditions hold, then an interface can be added to the architectural component implementing the super-feature to act as a switching mechanism between the respective sub-features' components. This is most frequently a variation of the Strategy design pattern [BJM^{+}95], where the `Context` is any calling component, the super-feature's component plays the role of the `Strategy` and defines an abstract interface, which is implemented by each of the sub-features' components that play the role of the `ConcreteStrategy`s.

NN-Trainer Examples

Deriving an interface from a uses interacts relation is always specific to the case at hand. For this reason, examples will be given from the NN-Trainer case study. In fig. 4.15, a partial view of the interacts and hierarchy relations for the training process of a NN was given. Throughout the transformation phase, these relations have been optimized. Fig. 4.37 shows a partial view of the FM after the Feature Interactions transformation phase.

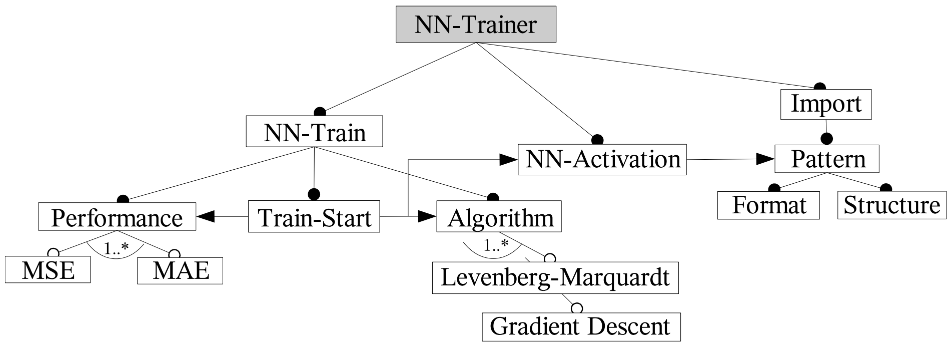


Figure 4.37: NN training related features after the third transformation phase

Based on the interacts and hierarchy relations, the PL developers derive interfaces for the architectural components that implement each feature. Note that before a training can be started, a NN must be designed with the **NN-Design** feature not shown in the figure and training patterns have to be imported with the **NN-Pattern** feature. Initially, the PL developers examine the interacts relation between the **Train-Start** and the **NN-Activation** features. The direction of the interacts relation indicates that the `Train-Start` component will require an interface from the `NN-Activation` component. The specification of this interacts relation leads to the addition of the interface of listing 4.2 in C++ notation.

Listing 4.2: `NN-Activation` interface

```
double Activate(u_long ulNNID, u_long ulPatternID)
```

This interface receives as input an identification number of the NN that must be activated and an identification number of the pattern to be used for the activation. The interface defines that the activation of the NN should be returned in the form of a double-precision number. The NN ID is used by the `NN-Activation` component to extract the proper NN parameters from the `NN-Design` component, which is not shown in the figure. With these parameters, the `NN-Activation` component can gather the needed structural information for the activation of the NN, e.g. number of neurons, NN-Architecture, weights, etc. The pattern identification number is used by the `NN-Activation` component to load the proper training pattern from the `Pattern` component and apply it to the NN.

The **Pattern** feature has an aggregation relation to its sub-features. As mentioned above, such hierarchy relations indicate that the derived architectural component should play the role of a facade for the components derived from the sub-features. In the NN-Trainer case study, the `Pattern` component is an example of this case. The PL developers identify the type of hierarchy relation as an aggregation interacts relation and add the interface of listing 4.3.

Listing 4.3: `Pattern` interface

```
void * GetPattern(u_long ulPatternID)
```

Through this interface, the retrieval of a pattern is considerably simplified. It allows the `NN-Activation` component to get a specific pattern without having to provide any additional parameters, like e.g. the path and name of the pattern or define its structure. The `Pattern` component receives as input the pattern identification number that is to be used for the activation of the NN and performs the communication with the `Format` and `Structure` components to load the proper pattern. This identification number has been defined during the importing of the pattern into the system, just before the start of the NN training. The `GetPattern` interface returns a pointer to void, which is then casted by the `NN-Activation` component according to the structural parameters of the NN.

The activation of the NN is returned eventually to the `Train-Start` component, which then propagates it to the `Performance` component for the calculation of the NN performance. The **Performance** feature hierarchy is a specialization hierarchy. The **MSE** and **MAE** features implement special performance algorithms. Additionally, more than one of the **Performance** feature's sub-features can be selected by a PL customer due to the `1..*` cardinality. These factors fulfil the three preconditions for the implementation of a switch mechanism for the `Performance` component.

Based on this, the PL developers apply the Strategy design pattern to the **Performance** feature hierarchy. The `Performance` feature becomes the `Strategy` component that defines the abstract interface for the calculation of the performance (listing 4.4, line 3). This interface receives the NN identification number and the current NN activation and returns the performance of the NN as a percentage. The NN identification number is used to store the various NN activations for calculating their average value. This interface is implemented in each of the sub-feature components of the **Performance** feature, namely, the `MSE` and `MAE` components. These components play the role of the concrete strategies in the Strategy design pattern. For the complete implementation of the design pattern, the `Performance` component receives another interface to set the active performance algorithm (listing 4.4, line 6). The desired performance algorithm is set through an enumeration during the configuration of the NN training parameters.

Listing 4.4: `Performance` interface

```
1 virtual u_short CalcPerformance(u_long ulNNID,
                                  double dActivation)=0
3

5 void SetPerformanceAlg(PERFORMANCE ePerformance)
```

After the calculation of the performance of the NN, this is propagated to the `Algorithm` component. The **Algorithm** feature hierarchy fulfils the same preconditions as the **Performance** feature hierarchy and can therefore be implemented respectively, i.e. with the Strategy design pattern. The abstract interface for the algorithm component is given in listing 4.5. The `Algorithm` component calculates the new weights and biases of the NN based on its identification number and performance.

Listing 4.5: `Algorithm` interface

```
virtual void CalcWeights(u_long ulNNID,
                         u_short usPerformance)=0
```

The main difference to the `Performance` component implementation is that the `Algorithm` component encapsulates common functionality for its sub-features. The `Algorithm` component provides a body for the `CalcWeights` interface, which can be explicitly inherited by the concrete algorithm components. The body implementation performs various calculations based on the topology of the NN that are needed by many training algorithms. If this functionality is desired, then it can be explicitly called by the concrete algorithm components.

4.6 Architecture Development

The development of the PL architecture is performed progressively throughout the FArM method. Fig. 4.1 illustrates exactly this by presenting the *Architecture Development* phase as a constant activity of medium intensity performed in each iteration. At every point in time during the various iterations, the PL architecture reflects the FM structure. More precisely, each of the architectural components is derived directly from the corresponding feature of the FM. For each of the features there exists exactly one architectural component that implements it. This fact assures a strong feature-architecture mapping.

Throughout the FArM transformations, the granularity of the software components is optimized and the interfaces of the components are defined. Eventually, the components are placed within an architectural context, e.g. through the application of an architectural style and each of the components is implemented. The component implementation may also lead to the addition of new interfaces or even back to one of the transformation phases, where new features may be added or existing features altered.

NAR & Quality Features

During the *NAR & Quality* transformation phase, the PL developers create a first coarse software architecture for the PL system. This is composed solely of the architectural components derived from the features that persist throughout the first transformation phase. During this transformation phase, there exist no relations between the software components. Furthermore, the component specification is identical to the specification of the feature it is implementing.

Most of the architectural components during the NAR & Quality transformation phase are closer to the customer perspective. This is due to the fact that they originate primarily from the initial FM. The latter is created with a domain analysis method that is independent from FArM. Most domain analysis methods give the customer perspective a rather high priority for the definition of features.

Nonetheless, this fact has also a positive side. Namely, the PL architecture can cope well with market changes. For instance, changing of an existing feature, e.g. an extension, can be readily supported because of the separation of concerns based on the customer perspective, i.e. features implemented in an architectural component. This is facilitated by the resolution of any NAR and quality feature, which leads to a FM containing solely functional features. These in turn have a higher probability to be directly implemented in an architectural component.

Architectural Requirements

Nevertheless, it may not always be possible to directly implement a functional feature into one architectural component. This issue is addressed, among others, during the *Architectural Requirements* transformation phase. During this transformation phase, the architectural requirements of the system are explicitly addressed. Issues like communication mechanisms, data management, etc., which are not visible to the customer perspective, are handled.

Primarily, the PL architects focus on the various architectural requirements of the PL system in order to directly resolve them, integrate them into existing functional features and/or derive new features. Additionally, the specification of existing functional features is considered in relation to architectural requirements, which may also lead to the creation of new functional features that can replace or complement the functionality of a feature that cannot be directly implemented into one architectural component.

In this transformation phase, the PL architecture is enriched with features derived primarily from an architectural perspective. This is a conscious decision in the FArM method made in order to achieve a balanced software architecture from both the customer and architectural perspectives of the system.

As a result, just before the beginning of the next transformation phase, the software architecture contains a fair mix of both customer and architecture related components. Additionally, the specification of all features in the FM and respectively of the derived architectural components has been significantly concretized and any ambiguities with respect to architectural implementation have been resolved.

Feature Interaction

During the iterations of the feature interaction transformation phase, the developers address the interactions between the features of the FM. These are identified, optimized and finally, interfaces are derived based on the interacts relations. The derived interfaces are placed directly into the components that implement a feature.

This process practically completes the specification of the components. During the previous transformation phases, the main responsibilities of the components were defined. In this transformation phase, the dependencies and communication needs between the components are also established.

Furthermore, this transformation phase leads to higher maintainability and flexibility on the architectural level. This is realized in terms of the optimization of the feature interaction. From the maintainability perspective, FArM strives to achieve a balance between the number of hierarchy and interacts relations. This is translated on the architectural level as an increase of the encapsulation and decoupling of components, e.g. through the implementation of the facade design pattern in super-feature components. Additionally, features with a large number of interacts relations are identified and, according to their future change probability, transformed, so as to localize possible changes.

From the flexibility point of view, FArM natively supports the direct implementation of switching mechanisms, e.g. through the Strategy design pattern within feature hierarchies. This is translated on the architectural level as the addition of variability-specific interfaces to the respective architectural components that implement the features.

The performance of the system architecture is also addressed during this transformation phase. Interacts relations between features are translated to direct calls between the respective architectural components. This mechanism can be utilized by the PL architects to optimize the performance of the system for time-critical use case scenarios. Again, interfaces are derived from such interactions and are added to the components.

With the aforementioned actions on the architectural level, the PL developers can adjust the architecture of the system to the desired level of maintainability versus performance or add overall flexibility to the system.

Architectural Context

At some point in time, the PL developers commit to a specific architectural context for the derived components. This point in time is most frequently during the first few iterations of the Feature Interaction transformation phase. The architectural context is specified in the majority of domains by committing to an *architectural style*, also known as an architectural pattern.

There exist numerous architectural styles [BJM$^+$95], each having a number of advantages and disadvantages regarding maintainability, flexibility, performance, etc. FArM has been developed with primer focus on maintainability and flexibility, followed by performance. This priorities have to do with the fact that FArM targets the development of software PLs. The latter exhibit high complexity, require a lot of variability and for some domains also high performance. For these reasons, FArM primarily supports architectural styles that possess such attributes.

Microkernel

One of the main architectural styles supported by FArM is the *Microkernel* architectural style [BJM+95]. This architectural style separates a minimal functional core from extended functionality and customer-specific parts. The Microkernel serves as a socket for the plugging of these extensions and the coordination of their collaboration. Fig. 4.38 illustrates the main components of a Microkernel architecture and the relations between them.

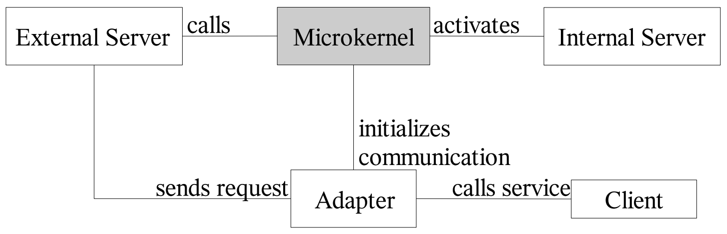


Figure 4.38: Microkernel architecture

In the Microkernel architectural pattern the `Microkernel` component provides core functionality, manages common resources, encapsulates system dependencies and offers the communication mechanisms between the various system components. The `Internal Server` components encapsulate system specific functionality and implement additional services. The `External Server` components provide services to `Client` components, either through their own implementation and/or through services provided by `Internal Server` components. `Client` components play the role of applications, which access `External Server` services through `Adapter` components. The latter hide system dependencies, such as the communication with `External Servers` for the access of services.

One main advantage of the Microkernel architecture is its extendibility. This advantage comes from the ability to plug new services into the `Microkernel` through the implementation of additional `External Server` or `Internal Server` components. The latter must comply to the communication protocol defined by the `Microkernel`. Afterwards, `Client` components can easily make use of this functionality. Furthermore, the Microkernel architecture illustrates enhanced flexibility. `External` or

`Internal Server` components can be added or removed from the system at compile or even at runtime, e.g. if they are implemented as DLLs (Dynamic Link Libraries).

The aforementioned attributes of the Microkernel architectural style readily suit the development of software PLs. On the one hand, PLs require high extendibility because of their long life-cycle. Software PLs represent a large investment that can provide significant gains when exploited over a long period of time. Over this period, new requirements are destined to arise from the domain, which must be rapidly satisfied by the PL in order to preserve its competitiveness in the market. This can be achieved through the implementation of new `External` and `Internal Server` components and their easy integration into the system.

On the other hand, flexibility is a primer concern in software PLs for the support of the variability of a domain. For instance, based on the PLs common core, a series of similar products must be instantiated. The instantiation process must be performed ideally with minimal effort. This is readily supported by the Microkernel architecture through the combination of the desired `External` and `Internal Server` components.

The FArM transformed FM and the derived components, along with their interactions can be directly mapped to the Microkernel architecture. Namely, the root feature of the transformed FM will be implemented in a component that will play the role of the `Microkernel`. Each of the derived components will either play the role of an `External Server`, an `Internal Server` or an `Adapter`. For a partial view of the NN-Trainer software architecture see Appendix C.

More precisely, the components derived from features that have been slightly changed throughout the transformation phases, will play the role of an `External Server`. This is suitable, because such components implement domain logic that is independent from system specifics. Components derived from features during the architecture requirements transformation phase are more likely to provide system specific services. Such components can readily play the role of an `Internal Server`. Components derived from FArM super-features are suitable candidates for the role of an `Adapter`. Super-feature derived components generically encapsulate the communication between features, which is exactly the role of an `Adapter` component in the Microkernel architecture. Finally, a `Client` component can be viewed merely as an implementation of a feature, which requests services from another feature.

The interacts relations of the FM can also be directly mapped to the Microkernel architecture. Since each feature of the transformed FM is implemented in one Microkernel component, the communication between the components reflects the interaction between the features. For example, a feature interacting with another feature through its super-feature is shown in the left part of fig. 4.39. The right part of the figure shows the mapping of the features to the Microkernel architecture. The root feature illustrated as a shaded box is implemented as the `Microkernel` component. Feature **A** is illustrated as a `Client` component, although it could be an `External` or `Internal Server` itself. Feature **B** is a super-feature and therefore is implemented as an `Adapter` component. Feature **C** is the feature that provides the functionality required by feature **A** and is thus implemented as an `External Server` component.

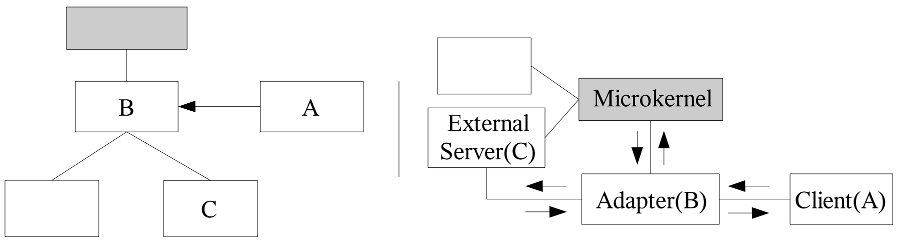


Figure 4.39: Mapping to the Microkernel architecture

The interaction between features **A** and **C** through feature **B** is translated into the following component communication:

1. The `Client` sends a request to the `Adapter`
2. The `Adapter` gets a reference to the `External Server` through the `Microkernel`
3. The `Adapter` propagates the request from the `Client` to the `External Server`
4. The `External Server` dispatches the request and returns the result to the `Adapter`, which in turn sends it back to the `Client`

This mapping of the FArM transformed FM to the Microkernel architecture further enhances the advantages inherited by the architectural style. FArM brings the extendibility and flexibility of the Microkernel architecture to the *feature level* and adds to the maintainability of the resulting system. Because of the one to one relation between the transformed FM and the architecture, features become first class entities. From the extendibility point of view, the system can be now directly extended in terms of features. New `External` or `Internal Server` components that implement the features can be easily plugged into the existing PL platform. The instantiation of PL products can be simple done through the selection of the desired features and their implementing components, i.e. `External` or `Internal Servers` and `Adapters`. The maintainability of the PL is also increased, since the features, which represent the main concerns in the PL, are effectively implemented either in one architectural component or in at most a few architectural components. Furthermore, the feature interaction is directly mapped to the component communication, allowing for a prediction of the impact a change would have on the system (see also chapter 5).

Other Architectural Styles

Despite the fact that the FArM method directly supports the Microkernel architectural style, yet other architectural styles can be used with FArM. For this purpose, the PL developers must map the derived architectural components to the entities of the chosen architectural style. This section will briefly discuss a few of the most widely used architectural styles, i.e. the *Layers*, *Blackboard*, *Broker* and *Model View Controller (MVC)* architectural styles.

The Layers architectural style decomposes a system in groups of subtasks, where each group is at a particular level of abstraction. Communication is allowed only between neighbor layers. The mapping of FArM derived components to the Layers architectural style can be achieved based on the FM hierarchy. Each layer comprises of the components derived from features belonging to the same hierarchy level as shown in fig. 4.40. The features of the same hierarchy level are most likely also at the same level of abstraction. If this is not the case, then the PL developers must adjust the FM hierarchy by going through the third FArM transformation phase. Another restriction that applies for the Layers architectural style is that no feature interaction is allowed between non-neighboring hierarchy levels. If such interacts relations exist, then the developers must repeat the third FArM transformation phase with the objective to merge these interacts relations into neighboring hierarchy relations.

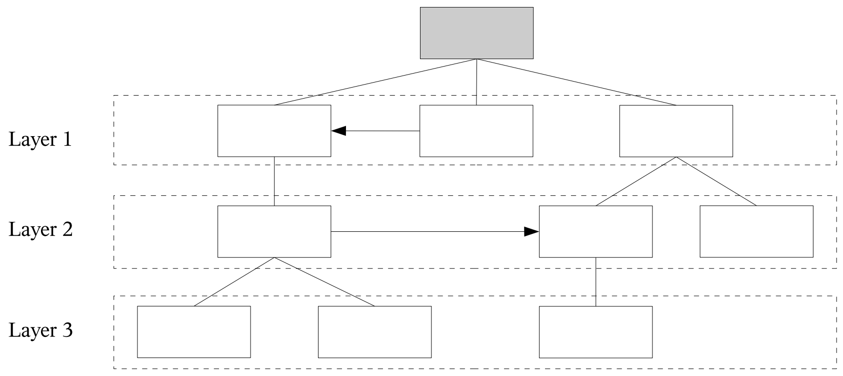


Figure 4.40: Mapping to the Layers architecture

The FArM transformed FM can also be directly mapped to an architecture adhering to the Blackboard architectural style. In a Blackboard architecture, the root feature will be implemented as the `Blackboard` component, while each of the features of the FM will take on the role of a `Knowledge Source` component. The `Control` entity of the Blackboard architectural style can be mapped in two ways. The first one is to delegate the responsibilities of the `Control` component to the `Blackboard` component. In this case, the `Blackboard` component will additionally decide which of the `Knowledge Source` components should be activated. If this approach is not satisfactory to the PL developers, then the second FArM transformation phase can be repeated for the identification of a feature or a set of features that are to take on the role of the `Control` component.

The main entities of the Broker architectural style require more compromises for their mapping to the FArM FM. Fig. 4.41 shows a possible mapping of a FM to a Broker architecture. The role of the `Broker` components is taken on by the components that implement the features having a direct hierarchy relation to the root feature. Each of their sub-features can be a `Client`, a `Server` or both. Interaction between the features is allowed only through the **Broker** features. Features not being under the same broker feature must send a request to the **Broker** feature of their hierarchy tree, which will then propagate the request to the proper feature through another **Broker**. This hierarchy and interaction enforced to the FM allows for a mapping to the Broker architecture. Each of the sub-trees of the FM can run on different network nodes. The communication between the `Broker` components will take place through remote data exchange, e.g. the http or ftp communication protocols, which also removes the need for `Bridge` components. `Client-side` and `Server-side Proxy` components are also not required, since the communication between components and the `Broker` will be derived from the hierarchy relations of the FM.

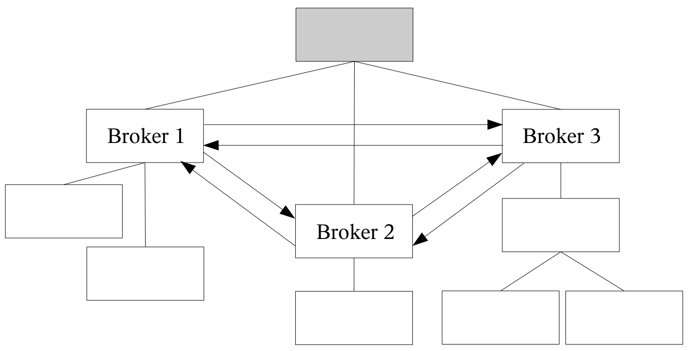


Figure 4.41: Mapping to the Broker architecture

Finally, the MVC architectural style entities can be mapped to the FArM features through categorization. The PL developers must identify features related to the presentation of data, the interpretation of data and the user interaction. The identified features must then be placed in the logical categories of the View, the Model and the Controller respectively. In a next step, the interactions between the features must be adjusted to conform to the MVC architectural style. Namely, the features belonging to the Model category must not use any features of the View category directly, rather indirectly. Each of the View features must provide a generic interface for communication with Model features, e.g. an `Update()` method that is to be triggered by the Model features when a change occurs that influences a View feature. The Controller features may directly interact with both the Model-related and View-related features to apply changes that occur through user interaction. Note that most FMs do contain MVC-related features. If no features of a category are found, then the PL developers must return to the FArM architecture requirements transformation phase and add new features that can take on the role of the missing category, e.g. Controller-related features.

4.7 Tool Support

A necessity for every methodology is tool support, i.e. a set of tools that will allow an efficient and consistent workflow with the method processes. FArM can be applied at the time of this writing with a set of industrial tools. These tools focus on the various phases of the method and can support all needs of the FArM developers.

One primary need of a FArM developer is to capture and manage feature specifications in the form of a FM. This process can be supported with a variety of industrial documentation management tools. An example is IBM's *Rational RequisitePro* [IBM06]. The hierarchy relations, cardinalities, etc. of a FM can be represented in the form of a structured list in such tools. Nevertheless, there exist numerous tools for the graphical representation and management of information, specific for feature modeling and PLs. Examples are the *XFeature* [PS06] and *DOME* [Hon06] tools.

FArM traceability links and interacts relations can be better captured and managed through the use of documentation management tools. The latter can also assist on finding interacts relations between features with the techniques described in section 4.5.1. For example, interacts relations can be identified through a recursive search through the feature specifications for the identification of common lexical structures, e.g. verbs, nouns, etc.

The modeling of the PL architecture can be performed with the use of industrial modeling tools, e.g. Borland's *Together* tool suite [Bor06]. The actual implementation of the PL components can be done on a development platform, e.g. with the Microsoft's Visual Studio [Mic06] tool suite.

For the deployment and instantiation of PL products a software dependencies and packaging tool can be used, e.g. the RPM Package Manager [Hat06]. Such tools are able to handle versioning issues and dependencies that may exist between FArM components.

Although the use of various tools for the implementation of the FArM method does provide efficiency regarding the workflow processes, it is rather hard to achieve consistency for the overall application of the method. For this reason, it would be advantageous to develop a unifying tool for the specific support of the FArM workflows. Such a tool has not been yet developed, although a large part of the tool specification has been made [Kau05]. In this student-work, it was identified that the optimum approach for a tool implementation would be a plug-in for the *Eclipse* platform [Fou06]. An implementation of such a tool is part of future work on FArM.

Chapter 5

Evaluation

This section will examine the extend up to which the goals set in section 2.6 have been achieved in this work. This includes at first an evaluation of the attributes of the methodology developed, i.e. of the FArM method. Afterwards, the strength of the mapping attained between features and the architecture will be evaluated. Then follows an evaluation of the efficiency with which feature-level variability is reached. Finally, this section will focus on the evaluation of PL product instantiation after the application of the FArM method.

5.1 Method Attributes

In section 2.6 the various attributes that the FArM method should possess were identified. These are, complying to a clearly defined methodical approach, the seamless integration into existing PL methods and generic support for currently used technologies and tools.

Methodical Approach

A vital precondition of the FArM method is its usability and comprehensibility by PL developers. This precondition is satisfied through a number of FArM attributes.

At first, the FArM method has been structured based on a broadly accepted development process, namely, an iterative process with clearly defined milestones. The FArM development process shown in fig. 4.1 is organized in four distinct transformation phases, which are completed in a series of iterations. On the one hand, this approach complies to widely accepted development standards in today's software industry and on the other hand, it is similar to other broadly used development methodologies, e.g. the Rational Unified Process. These factors assist the PL developers to adopt the FArM development method and integrating it into their own development processes. Furthermore, the FArM development process inherits the advantages of iterative development, e.g. early identification of risks and efficient distribution of project resources.

Another FArM attribute that contributes to its useability and comprehensibility, is the clear definition of the FArM phases. The FArM method has been designed to lead the PL developers with a series of distinct transformation phases to the desired results. Arguments for this are the small number of the FArM phases, namely, four, along with clearly defined pre and post-conditions for every phase. Additionally, within these phases, the FArM method clearly defines steps that assist the PL developers to achieve the desired post-conditions with high probability, regardless of the given domain. For instance, the third transformation phase clearly defines as a pre-condition the existence of exclusively functional features originating from both the problem and solution domain. The post-condition is the derivation of architectural components along with their interfaces for each of the features of the transformed FM. Within this transformation phase, the PL developers are assisted through a series of distinct steps, i.e. Identification, Optimization and Interface Derivation.

Finally, the useability and comprehensibility of the FArM method is supported by the produced literature of its application within various domains. The FArM method has been applied to the domain of Integrated Development Environments (IDEs) [Soc04], [SRP04], the domain of mobile phones [Kau05], [SRP05], [SRP06] and the domain of artificial Neural Networks presented in this work.

PL Method Integration

It was identified that the method should be able to seamlessly integrate into existing PL development methods. This attribute of the FArM method comes from the need to utilize existing knowledge and experience in the development of PLs that has been acquired from other PL methods. Furthermore, this integration naturally increases the useability and acceptance of FArM.

FArM can be integrated with existing PL methods at the transition point from the domain analysis to the architecture development. For example, the FArM method can be integrated into the FeatuRSEB method. Right after the development of the initial FM, the FArM method can be applied replacing the existing FeatuRSEB processes where appropriate. The derived architectural components can then be used in the Layers architectural style that is used in the FeatuRSEB method [GAd98] as described in section 4.6.

FArM can also be readily integrated into PL methods that place their focus either on the domain analysis or after the architecture development processes. An example of the former case is the use of the FODA method for the creation of the FArM initial FM. This can then be directly used for the application of FArM. An example of the latter case is the use of the Hyperspace approach with FArM. In this case, the PL components can be implemented with the FArM method and then each one of the components can be modeled as a hyperslice. With this approach, the PL developers can have the advantages offered by the Hyperspace approach, as well as the advantages of the FArM method.

Technology Support

The FArM method must also be compatible with the technologies currently used in software development, e.g. object-orientation, architectural and design patterns and tools. This is a vital precondition for the efficient application of the method.

FArM can be used in combination with any programming language, e.g. object-oriented or procedural. This is evident by the fact that FArM models the architectural components of the PL, but does not impose any restrictions regarding their implementation. The latter can be done with any programming language or platform that satisfies the needs of the PL domain.

FArM also explicitly supports and encourages the use of architectural and design patterns. Design patterns may be used in each iteration of the FArM method during the architecture development phase. These can be used, e.g. for the internal implementation of the FArM components or for the implementation of variability mechanisms, as described in the cases of components derived from super-features (sect. 4.5.3). Architectural patterns are also applied in FArM, usually during the third transformation phase. FArM primarily supports the application of the Microkernel architectural pattern. An extensive discussion of the application of this and other architectural patterns in FArM is given in section 4.6.

Tool support is also a very important issue that is addressed in FArM. The FArM developer can use a number of industrially available tools for the application of FArM. These range from documentation management tools, to development platforms and packaging tools. Nonetheless, as already mentioned in section 4.7, the development of a FArM-specific tool would be beneficial to the FArM developers.

5.2 Feature-Architecture Mapping

The main goal of this work is to provide a stronger mapping between features and the architecture in the context of PLs. Section 2.6 discussed the specific requirements for this mapping in relation to the state of the art methods' problems. The results of this discussion pointed out the need for a mapping that allows the application logic of one feature to be implemented into an architectural component and the feature interaction to be reflected by the component communication. Additionally, it was identified that solution domain entities must sometimes participate on the design of the PL architecture and should therefore be explicitly considered. Finally, it was pointed out that the actual mapping mechanisms must resolve the problems evident in the state of the art methods, i.e. the excessive use of traces in FeatuRSEB and the introduction of extra constructs in the generative programming technics, e.g. hyperslices in the Hyperspace approach. The following sections will look into the aforementioned objectives and evaluate the FArM approach from the feature-architecture mapping perspective.

Application Logic Mapping

At first, the extend up to which the application logic of a feature is actually implemented into an architectural component will be examined. FArM utilizes for this purpose an initial FM, developed with a domain analysis method. The FArM developers are free to select any domain analysis method that suits their needs, e.g. FODA. No assumptions are made regarding the nature or hierarchy of the features of the initial FM. Afterwards, in three distinct transformation phases, FArM transforms the features of the initial FM in order to derive the PL architectural components.

The application logic of a feature is defined by the specification given in the initial FM. Throughout the FArM transformation phases, any of the FArM elementary transformations may occur (sect. 4.2). That is, either the whole feature or a part of its specification may be directly transformed or merged with another feature or new features may be created to implement a feature's specification. The FArM elementary transformations can therefore cause the initially defined application logic of a feature to be eventually mapped to either none or more than one feature.

Because of the fact that exactly one architectural component is derived for each feature of the final transformed FM, the application logic of an initial feature will be implemented respectively into none or more than one architectural components.

On the one hand, if a feature is entirely transformed with the direct elementary transformation, then it is practically not present in the transformed FM. This implies that the feature will not be implemented in an architectural component. Nonetheless, FArM allows such a transformation to take place only for features that have minimal effect on the software architecture. These are the so called NAR (Non-Architecture Related) FArM features, which are handled in the first transformation phase (sect. 4.3). On the other hand, the direct elementary transformation may also take place only for parts of a feature's specification. Similarly, these specification parts should have no impact on the software architecture and thus can be resolved with alternative approaches, e.g. managerial solutions. Therefore, the "loss" of such application logic has no effect on the quality of the PL architecture.

With any other of the FArM elementary transformations the application logic of a feature can be mapped into more than one feature and respectively may be implemented into more than one architectural component. Nevertheless, the number of components into which a feature may be eventually implemented is constrained to a minimum in FArM. In most cases, a feature needs to be implemented into at most a few components. An example from the NN-Trainer case study is the **Neural Networks** feature (fig. 4.31), which is transformed with the create elementary transformation and is eventually implemented into the **NN-Design** and **NN-Train** features (fig. 4.32).

There may of course exist cases where a feature is mapped to many features of the FM, i.e. it is merged with numerous features or many new features are created to implement the feature. Such transformations occur primarily on quality features. In the majority of cases, such mappings cannot be avoided due to the broad impact that quality features have on a software system. An example of such a case is the transformation of the **Efficiency** quality feature (sect. 4.3.2). This quality feature had to be merged with a large number of functional features in order to be mapped to the architecture. Nevertheless, FArM still provides for a higher feature-architecture mapping compared to the contemporary methods. The merge of quality features occurs either into pre-existing functional features or into new functional features. The mapping of functional features to a large number of features is rather seldom in FArM and occurs mainly in time-critical domains for the enhancement of performance.

Another factor that enhances the mapping of features to the architecture in FArM is the fact that the elementary transformations are only applied when absolutely necessary. That is, FArM gives high priority to the preservation of the conceptual integrity and strives to maintain a direct mapping between features and the architecture. An example can be found in the case of the **License** feature (sect. 4.5). Although this feature interacts with a large number of features, it is not merged with each one of them in order to preserve the system's conceptual integrity.

From the above discussion it becomes obvious that the FArM elementary transformations cause in the majority of cases minimal scattering of application logic. In the cases where a feature is mapped to a large number of other features, FArM constraints the scattering of the application logic on the feature level, introducing no solution domain entities that drastically weaken the feature-architecture mapping. Finally, no tangling of a feature's application logic occurs in FArM, since each feature of the transformed FM is implemented into exactly one architectural component.

Feature Interaction Mapping

Another decisive factor for the achievement of a stronger mapping between features and the architecture is to allow for the component communication to reflect the feature interaction. When the application logic of a feature is mapped to at most a few architectural components, it is of great advantage, when also the feature interaction can be mapped to the component communication. This allows consequently for a stronger feature-architecture mapping.

This is achieved in FArM through an explicit transformation phase based on feature interaction (sect. 4.5). In this transformation phase, all possible interacts relations are identified and transformed to *uses* interacts relations. Afterwards, an optimization of these interacts relations takes place. The main goal of these optimizations is an increase of system maintainability through the transformation of interacts relations to hierarchy relations, which in turn leads to an increase of encapsulation and decoupling. Additionally, a normalization of the number of interacts relations takes place to further minimize the impact that future changes can have on the system. Finally, based on the optimized *uses* interacts relations between the features of the transformed FM, interfaces for the respective architectural components are directly derived.

A concrete example of the results of this FArM transformation phase is given at the end of section 4.5, where the interfaces of the components participating in the training of a NN are derived. This example reveals the direct mapping of feature interaction to component communication. The domain specific interaction scenario between features can now be directly mapped to component communication through methods required and provided by the architectural components, e.g. the `CalcWeights()` method is implemented in a concrete `Algorithm` component, which reflects the interaction between the **NN-Train** and **Algorithm** features.

With this process, FArM directly utilizes feature interactions to derive the requires and provides component interfaces. Furthermore, FArM performs explicit steps for the optimization of these interactions, which also have a further positive effect on the system maintainability.

Solution Domain Entities

During the exploration of the state of the art methods for the development of PLs, the need for the consideration of solution domain entities in relation to the design of the PL architecture was identified. Namely, it has been shown that the utilization of solution domain specific entities in the software architecture is sometimes indispensable for the enhancement of system maintainability, performance, etc. A characteristic example is the instantiation of FAD archetypes based on solution domain entities (sect. 2.4.3). It is thus of great importance for the FArM method to provide a balance between problem domain and solution domain entities in the PL architecture. The former is performed through the derivation of architectural components based on PL features, while the latter is explicitly performed in FArM's architectural requirements transformation phase.

The second FArM transformation phase focuses on the architectural requirements placed upon the PL. The PL developers identify and handle the architectural requirements of the system with the FArM elementary transformations. In this transformation phase, besides the direct resolution and merging of architectural requirements into pre-existing functional features, it is very likely that new features are added to the FM to satisfy the architectural requirements. These features are mainly conceptualized from the PL architects, but must also be approved by the feature analysts. The approval of features originating from the solution domain indicates that the features are on one hand understood by the feature analysts and on the other hand that they are of importance for the majority of the PL stakeholders. This fact justifies the introduction of these features into the FM.

A representative example of this case from the NN-Trainer case study is the introduction of the **Network** feature (sect. 4.4). The **Network** feature enables the training of NNs in a computer network. This feature effectively satisfies the architectural requirement to train large NNs with specific training algorithms that consume significant amounts of memory. The responsibilities of this new feature are mainly providing the communication and synchronization mechanisms between the various NN-Trainer instances that run on the different network nodes. The **Network** feature is explicitly approved by the feature analysts before it is introduced into the FM. Its eventual introduction into the FM is justified by the fact that most of the PL customers possess a computer network and are thus familiar with the concept of distributed applications. Furthermore, it is of great significance for the prevalence of the NN-Training in the NN market to enable the training of large NNs, even with training algorithms that have high memory requirements.

The above discussion illustrated that FArM explicitly introduces solution domain entities into the PL architecture through the handling of architectural requirements. Additionally, the process with which these solution domain entities are introduced is elevated to the feature level. This fact is consistent with and supports a stronger feature-architecture mapping.

Mapping mechanism

Many of the state of the art PL methods recognize the need for a strong mapping between features and the architecture. This is also evident by the fact that a number of these methods have introduced certain mechanisms to realize this mapping. Representative examples are FeatuRSEB's *traces* (sect. 2.3.2) and the *hyperslices* constructs of the Hyperspace approach (sect. 2.5.1). Nevertheless, these mechanisms operate on architectures with a high scattering and tangling of features. This leads in the case of the FeatuRSEB method to an explosion of the number of traces needed for the mapping of features to the architecture. In the case of the Hyperspace approach, hyperslices are hard to create and maintain, while providing a superficial separation of concerns, evident by the hyperslice interaction. One of the goals of this work is to provide an efficient mapping mechanism between features and the architecture.

FArM makes use of traceability links for this purpose (sect. 4.2). FArM traceability links are created between the transformed features and the features of the transformed FM that took part in the transformation. If the direct elementary transformation is applied, then a traceability link is added between the transformed feature and the root feature of the transformed FM. In the case of the merge and create elementary transformations, traceability links are created between the transformed features and the features in the transformed FM with which the feature was merged or in which the feature was implemented.

The FArM traceability link has a double role. The traceability link provides a mechanism to follow the transformations that occur on a feature from the initial FM, down to the final transformed FM and the respective architectural component(s). This link serves for forward and backward traceability. Furthermore, the traceability link holds the rationale of the transformations on the feature, i.e. the reasons and thoughts behind the decision to transform the feature. This information is invaluable for the maintenance of the system. A FArM traceability link has been expressed in the form of XML code in [Kau05].

An example of a FArM traceability link for the **Neural Network** feature is illustrated in listing 5.1. As shown in the listing, each traceability link is assigned an identification number (`id`). Additionally, every feature receives an id, which is different between the various versions of the FM, i.e. the same feature receives a new id after each transformation phase. Within the `<tphase>` tag the transformation phase is defined. This also sets the FM in which the origin of the traceability link is situated. The `<sourcefeature>` tag defines the feature that is transformed. The `<targetfeature>` tag defines the feature that takes part in the transformation. The `<telementary>` tag defines the FArM elementary transformation that is applied on the source feature and contains the rationale of the transformation. For the direct elementary transformation no target feature is defined. For each elementary transformation, a new traceability link is created between the features of the FM before and after the transformation.

Listing 5.1: XML notation of FArM Traceability Links

```
<tracelink id="123">
  <tphase>3</tphase> <!-- 3 = Feature Interaction -->
  <sourcefeature id=15>
    <name>Neural Network</name>
  </sourcefeature>
  <targetfeature id=57>
    <name>NN-Train</name>
  </targetfeature>
  <telementary name=Create>
    <rationale>
      The NN-Train feature takes on the responsibility
      of training a NN. It coordinates the NN training
```

```
        by interacting with other features.
      </rationale>
    </telementary>
        ...
</tracelink>
```

The main advantage of FArM traceability links in comparison to the existing mechanisms is the fact that a finite number of traceability links is required to map a feature to the architecture. This is due to the small number of transformations that occur in average during the FArM transformation phases and the one to one relation between features of the final transformed FM and architectural components. Furthermore, the creation and maintenance of FArM traceability links requires far less effort compared to the hyperslice mechanism of the Hyperspace approach. This effort can be further reduced, e.g. through tool support and the use of the XML format given in listing 5.1.

Maintainability

One of the main advantages of the stronger feature-architecture mapping achieved in FArM is system maintainability. High system maintainability is accomplished when changes can be quickly performed and require small effort. The most crucial factor for achieving high maintainability is the locality of change, i.e. the extend up to which a change propagates into the system. In order to minimize the impact of changes, the system must illustrate a suitable separation of concerns. That is, the main concerns of the system, which are also most likely to be modified in the future, must be as much encapsulated and decoupled as possible. In the context of PLs, this separation of concerns must be performed on the basis of features. Ideally one feature should be implemented into exactly one architectural component. Although this is not always possible, the PL architecture must at least illustrate a strong mapping between features and the architecture.

FArM provides this stronger mapping between features and the architecture in a number of ways. On the one hand, through the encapsulation of the application logic of a feature into at most a few architectural components, through the mapping of the feature interaction onto the component communication and by providing an efficient traceability mechanism from features of the initial FM to architectural components. On the other hand, FArM incorporates solution domain entities into the PL architecture design, while assuring that these entities are compatible with the existing PL features. It also performs an optimization of the feature interaction to enhance the encapsulation and decoupling between features. Eventually, for each of the features of the final transformed FM exactly one architectural component is derived. All these processes allow for a stronger mapping between features and the architecture in comparison to the state of the art PL methods.

FArM is also compatible with numerous architectural styles for the concrete implementation of the derived components. The Microkernel architectural style is especially supported by FArM (sect. 4.6). With this architectural style, the FArM developers can take full advantage of the feature-architecture mapping provided by FArM. Furthermore, each feature can be directly mapped to a Microkernel plugin component with specific requires and provides interfaces, as defined by the uses interacts relations of the features.

The advantages of a stronger feature-architecture mapping regarding maintainability can also be shown in the context of a concrete example from the NN-Trainer case study. Figure 4.25 shows the **Pattern** feature hierarchy. The responsibility of this feature is to import patterns for the training of a NN. These can have various formats, e.g. binary, text, etc. and various structures, e.g. they may consist of elements separated by semicolons, spaces, commas or a combination of both. The imported patterns are then made available to the various features of the FM for the training of NNs. Listing 4.3 shows one of the interfaces of the `Pattern` derived component for the retrieval of a pattern. This interface receives the identification number of a previously loaded pattern and returns the pattern as a void pointer. The pointer is then casted to the proper format needed based on the structure of the NN to be trained. Listing 5.2 shows the interface for importing a pattern. This is the interface that imports the pattern into the system given a file path. Upon a call to this interface, the given file is scanned for the identification of its format and structure and a unique ID for later access to the pattern is returned.

Listing 5.2: `GetPattern()` interface

```
u_long GetPattern(char* path)
```

For comparison, we assume a software architecture of the NN-Trainer system, which has been designed with conventional use-case-oriented techniques, e.g. as would be the case with the FeatuRSEB method. In this software architecture, each particular component has knowledge of and handles the loading of patterns. Each component needs to load patterns with specific formats and structures, e.g. images, film streams, etc. The following change scenario then occurs:

"A new image format shall be supported by the system for the solution of pattern recognition problems with the following format: ... and structure: ..."

In the hypothetical NN-Trainer system designed with the FeatuRSEB method, all components related to pattern recognition are affected by the aforementioned requirement. Each one of the components must be internally changed to support the new image format. For the FArM developed NN-Trainer system, this change remains local. New features must be added under the **Format** and **Structure** features to support the format and structure of the new image format. The respective architectural components must also be derived and plugged into the Microkernel architecture. All other system components remain unaffected by this change and may continue to use the `GetPattern()` method of the **Pattern** feature exactly as before.

This change scenario illustrates the advantages of a stronger feature-architecture mapping. The effects of feature scattering and tangling are minimized in the FArM architecture, while the hypothetical conventional architecture suffers from propagating changes. In the hypothetical architecture the importing of a pattern, which is actually a main concern of the system and thus a feature, is unavoidably implemented throughout the system because of the use-case-oriented development process. FArM on the other hand identified the importance of this concern as a system feature and encapsulated it into one loosely coupled component.

The problems occurring from this change scenario might have been predicted by an experienced architect of the hypothetical architecture, who would then build a layer to encapsulate the pattern importing functionality. Nonetheless, this would doubtably be consistently performed for each feature of the system. FArM provides a methodical, structured approach for the encapsulation of the main system concerns through the enhancement of the feature-architecture mapping.

Scattering & Tangling in the NN-Trainer PL

In order to provide an indication of the ability of the FArM method to limit feature scattering and tangling, this section will provide a few numbers regarding these two issues drawn from the NN-Trainer PL case study. Figures 5.1 and 5.2 show the distribution of scattering and tangling in the NN-Trainer PL.

One can read the scattering histogram of fig. 5.1 as follows:

"$< y >$ *feature(s) is/are implemented in* $< x >$ *component(s)*"

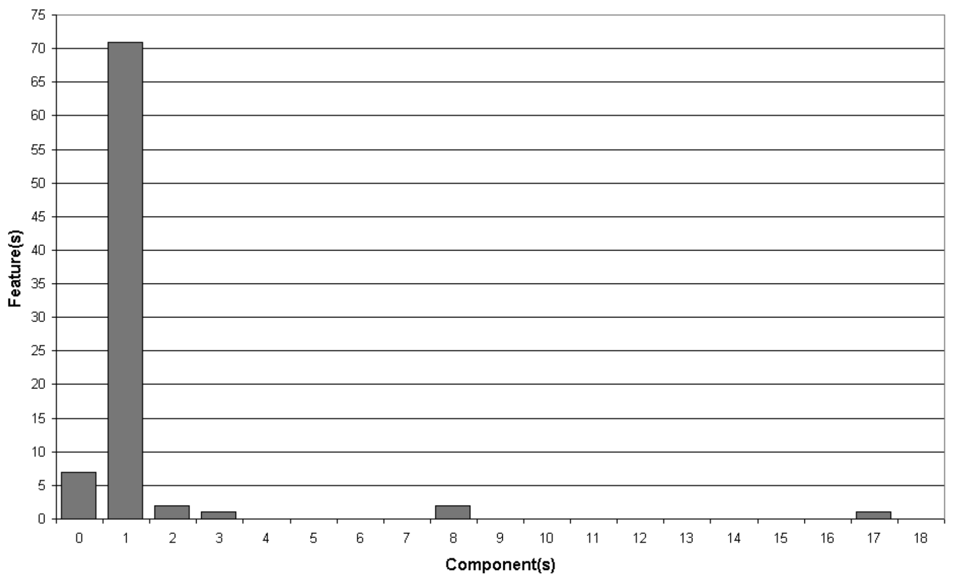


Figure 5.1: Histogram of scattering in the NN-Trainer PL

Note that when referring to the implementation of a feature in an architectural component, it is actually referred to the implementation of the feature's application logic. That is, a component implementing a feature can have access to other components' functionality, but it has no knowledge of the internal implementation of this functionality. Furthermore, the features taken into consideration for the construction of the aforementioned diagrams are the features of the initial FM.

From the histogram of fig. 5.1 it can be concluded that 71 features of the initial FM are implemented in exactly 1 architectural component, i.e. for 71 features of the initial FM there exists no scattering. Since there exist overall 85 features in the initial FM, in approximately 92% of the cases there was no scattering. Merely 8% of the initial features was scattered in the NN-Trainer PL.

As shown in the left-most bar of fig. 5.1, 7 features of the initial FM needed no implementation within an architectural component. These are the NAR features, which were directly resolved in FArM. A few functional features, along with the quality features caused the deviation seen in the scattering histogram. For instance, the **Recoverability** quality feature was implemented into 3 different architectural components (sect. 4.3.2). The feature most scattered in the NN-Trainer PL was the **UI** feature and its **GUI** and **Command-Line** sub-features. This is indicated by the right-most bar of the scattering histogram. Nonetheless, this has been a conscious decision of the PL developers, since there is little change probability for these features in the future. The developers decided to provide a stable interface to the users of the NN-Trainer products, so as to reduce the learning curve for future versions of the software.

The histogram for the tangling occurring in the NN-Trainer case study can be seen in fig. 5.2. This diagram can be read as follows:

"$< y >$ *component(s) implement* $< x >$ *feature(s)*"

Again, implementation refers to the application logic of a feature of the initial FM.

The histogram shown in fig. 5.2 reveals that 57 components implement the logic of exactly one feature. Since there are 87 components in total, approximately 65% of components contain no tangling in the final NN-Trainer software architecture. Moreover, at most 3 features are implemented into an architectural component. Note also that a significant part of the tangling in the NN-Trainer software architecture is due to quality features, e.g. **Efficiency** (sect. 4.3.2), which force their implementation in numerous functional features, thus increasing feature tangling.

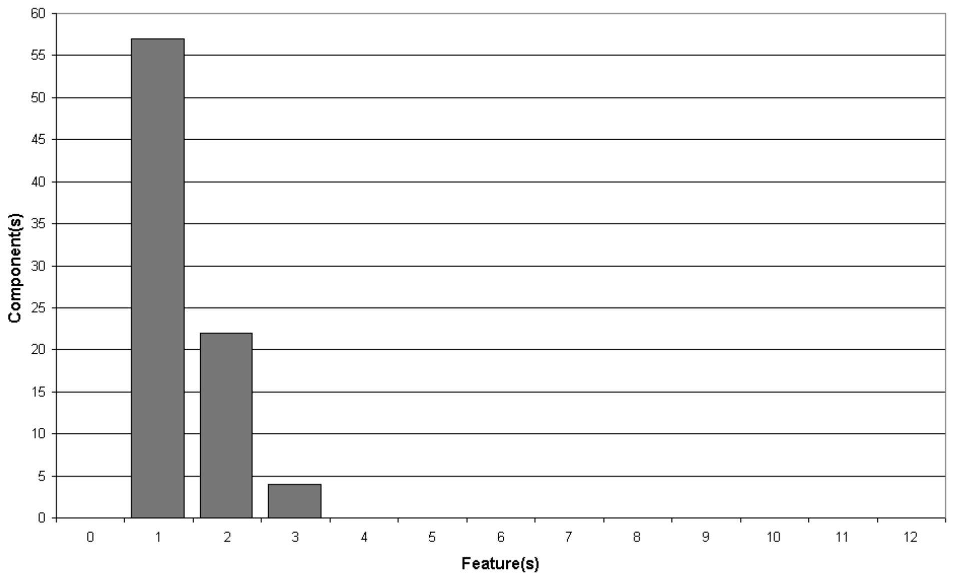


Figure 5.2: Histogram of tangling in the NN-Trainer PL

It can be concluded that for the NN-Trainer case-study most features where implemented into exactly one architectural component, while a large percent of components illustrated no tangling. In the cases where tangling did occur, it was limited to the tangling of at most 3 features in a component. The above discussion indicates that FArM does indeed minimize feature scattering, while significantly limiting feature tangling.

5.3 Feature-Level Variability

Another primary goal of this work is to provide efficient variability on the feature level. Achieving efficient variability of features enables maximum gains from adopting a PL approach. This is the case, because the various PL products can be naturally expressed as a set of features. Varying feature constellations will consequently yield various products. An efficient feature-level variability minimizes the effort needed for instantiating these feature combinations, thus fully exploiting the possibilities of the PL and maximizing the return on the initial development investment.

Section 2.3.3 presented the various variability mechanisms applied in the state of the art PL methods and provided an evaluation of these mechanisms with respect to their application on the feature level. The following sections will illustrate, how these problems are resolved in combination with the FArM method.

Feature Tangling

One of the problems of numerous variability mechanisms regarding their efficient application on the feature level is feature tangling. Examples of such variability mechanisms are the "null" component variability mechanism, linking, configuration management and code fragment superimposition, e.g. the Hyperspace approach. These variability mechanisms can only operate efficiently on the feature level if features are only scattered throughout the system, but not tangled with other features.

FArM minimizes feature tangling in the PL architecture. Features may indeed use other features, but this is controlled by the allowed interacts relations between the features. The latter have been further explicitly optimized in FArM, both with respect to quantity and direction (sect. 4.5.2). Additionally, each feature is implemented in either one or at most a few different architectural components. Therefore, feature scattering is also minimized in FArM. Nonetheless, a certain degree of feature scattering may occur for a number of features. Despite this fact, the above variability mechanisms can be applied more efficiently in combination with FArM.

An example can be seen in the application of the configuration management variability mechanism. If the features of the transformed FM are directly mapped to an architectural component, then the configuration management tool can be efficiently used for the instantiation of a product based on a set of features. Each of the features of the initial FM can be programmed into the configuration management tool along with the traceability links to the transformed FM and eventually, to the respective architectural components. Features of the initial FM can then be selected by the customer and the corresponding architectural components can be automatically chosen by the configuration management tool. If the customer has no special customization needs, the final product can also be automatically deployed.

Precondition Enforcement

A number of variability mechanisms enforce certain rigid preconditions for their application on the feature level. Two representative examples of such variability mechanisms are infrastructure-centered architectures, e.g. CORBA, COM, etc. and ADLs.

Through the use of FArM, the PL developers are more flexible in the selection of variability mechanisms. With FArM, other variability mechanisms that do not enforce any special preconditions can be applied, instead of the aforementioned mechanisms. For instance, one may use a configuration management tool, instead, of an ADL to achieve alternative variability during product architecture derivation.

Furthermore, FArM does enforce the precondition of following the FArM development processes, but it still remains more flexible than the aforementioned variability mechanisms. For example, the architecture development of FArM can be performed with any architectural style, e.g. Microkernel, Layers, Broker, etc., rather with e.g. a CORBA architecture. Additionally, the PL developers are unhindered in the selection of an implementation technology in FArM, in contrast to an infrastructure-centered architecture or an ADL approach.

Feature Size

Yet another problem identified for the application of the contemporary variability mechanisms on the feature level is that a number of variability mechanisms can only be efficiently applied for small-sized features. Examples of such variability mechanisms are condition on constant, condition on variable and design patterns, e.g. strategy. This is mainly due to the fact that features are scattered and tangled throughout numerous architectural components. Therefore, the PL developers must invest a lot of effort to first identify a feature's various parts and then repeatedly apply the aforementioned variability mechanisms throughout.

With the stronger feature-architecture mapping provided by FArM, this issue is resolved. The minimization of feature scattering and tangling, as well as the elevation of features to first class entities in the architecture, lead to the immediate application of the aforementioned variability mechanisms on the feature level.

An example can be shown with the application of the Strategy design pattern on the feature level. This can be seen in listing 4.4, which shows the interfaces belonging to the features of the **Performance** feature hierarchy. In this example, the component derived from the **Performance** feature, which is the super-feature in a specialization hierarchy relation, receives the `CalcPerformance()` and `SetPerformanceAlg()` interfaces and plays the role of the abstract *Strategy* object. The components derived from the sub-features **MSE** and **MAE**, provide an implementation for the `CalcPerformance()` interface and play the role of a concrete Strategy object.

As shown in this example, the Strategy design pattern is slightly changed for its application on the feature level. Namely, the interfaces mentioned above must be placed in facade classes, since the `Performance`, `MSE` and `MAE` components consist of numerous classes. Additionally, setting the current performance algorithm is not made in a `Context` object, as in the original design pattern, rather directly in the *Strategy* object, i.e. the facade class of the `Performance` component. Nevertheless, the essence of the Strategy design pattern is present in the aforementioned constellation, allowing for the direct and efficient application of the pattern on the feature level.

Inhomogeneous Entities

One problem that becomes evident for variability mechanism when applied on the feature level is that they all introduce numerous inhomogeneous entities, which decrease the system's conceptual integrity. Examples of such variability mechanisms are condition on constant, condition on variable, design patterns, code-fragment superimposition, etc. After the use of FArM, the same variability mechanisms can be applied more efficiently on the feature level, with minimal introduction of extra entities.

For instance, the Hyperspace approach can be used much more efficiently after the application of the FArM method. Namely, hyperslices can be effortlessly defined to include only the feature-related code derived from the FArM final FM. Because of the upfront separation of concerns, the combination of hyperslices does not require any special compositional rules, rather only the Hyperspace *merge* rule, since feature tangling is minimized.

Unpredictable Effort

Lastly, all contemporary variability mechanisms require an unpredictable amount of effort for their application on the feature level in conventional PLs. This is due to the fact that there is no knowledge of the degree of feature scattering and tangling in advance. A feature may be scattered throughout the system, while it can be intensively tangled with other features. This makes it very difficult to predict the effort needed for the application of a variability mechanism on the feature level.

This problem is resolved in FArM through the stronger feature-architecture mapping and the consistent creation of traceability links and interacts relations. Because of the minimization of feature scattering and tangling, a feature can be easily localized to at most a few architectural components. Furthermore, there exists a clear mapping between features of the initial FM and the architectural components through FArM traceability links. Finally, the implications of applying any of the variability mechanisms to these components can be foreseen based on the interacts relations. The latter point out which features are influenced by the application of the variability mechanisms, thus enabling a precise estimation of the effort needed.

5.4 Product Instantiation

Another main goal of this work is to enable a generative approach to product instantiation, which does not suffer from the problems identified for the generator-based framework component model (sect. 2.4.6).

Enabling a generative approach to product instantiation is generically achieved in FArM through the stronger feature-architecture mapping and the efficient application of variability mechanisms. Each feature of the initial FM can be mapped to at most a few architectural components, upon which numerous variability mechanisms can be efficiently applied, e.g. design patterns. Feature tangling is also minimized in FArM. These factors generically enable for a generative product instantiation. If additionally the final FArM PL architecture is constructed with the Microkernel architectural style (sect. 4.6), then the instantiation of PL products based on features is reduced to the plugging of the right components into the Microkernel component. In the cases where another architectural style is chosen for the implementation of the PL architecture, then a generative approach to product instantiation can still be achieved through the application of a suitable variability mechanism. Because of the strong feature-architecture mapping, the complexity of applying a variability mechanism, e.g. the Hyperspace approach, is minimized.

Nonetheless, the generator-based framework component model has been identified as a representative state of the art approach to generative product instantiation. In order to evaluate the extend up to which FArM has improved the generative product instantiation in comparison to this model, the following sections will go into each of the identified problems of the model and show how these are resolved in FArM.

Mature Domain

One of the problems of the generator-based framework component model is the need for a mature domain. The model requires a well-known domain for the identification of the right fine-grained extension components and variation points. FArM on the contrary can be used with any domain, regardless of the degree of knowledge the PL developers have of the domain. This is on the one hand due to the initial explicit domain analysis performed in FArM and on the other hand due to the fact that FArM is entirely based on features for the derivation of architectural components and variation points.

Just before the application of FArM, an analysis of the domain is required for the creation of the initial FM. This can be done for example with the FODA method. The resulting initial FM is then used as input to the FArM method. For each feature of the initial FM, an architectural component is derived. This dramatically simplifies the architecture development, regardless of the domain knowledge of the PL architects. Additionally, the PL developers are guided by the FArM phases for the refinement of the PL architecture and the definition of the right component granularity. The initial features are transformed and new features may be created throughout the FArM transformation in a methodical way, so that the final transformed FM has the proper granularity for the direct derivation of architectural components. Because of the strong mapping between features and the architecture, the variation points, which are generically present in the FM, can also be directly mirrored onto the PL architecture. This allows for the application of FArM in domains where little or no experience is present by the PL developers.

Diminished Evolvability

Another problem identified for the generator-based model is the constant need for change of the configuration tool or DSL during respective changes to the PL framework. These changes lead to extra effort from the developer point of view to evolve and maintain the tools and DSL. This issue can cause very high costs to a company that has based its product instantiation on such an approach.

In the case where the PL architecture is developed with the Microkernel architectural style, as suggested in FArM, there is no need for extra tools or a DSL for product instantiation. The PL architecture generically provides the needed mechanisms for a generative product instantiation. In the cases where another architectural style is chosen by the PL developers, then the tool configuration and evolution is much simpler in comparison to a system constructed with the contemporary PL methods. This is due to the stronger mapping between features and the architecture achieved in FArM.

For example, the addition of a new feature in a FArM architecture would be performed with the following procedure: First, the feature would be placed into the initial FM and the various relations between the feature and the other features would be created, i.e. hierarchy, requires and excludes relations. Afterwards, the feature would go through the FArM transformation phases. Finally, the entire feature or parts of it would be either merged with other features or new features would be created to implement the feature. Based on the transformation results, one or more architectural components would be derived to implement the feature. The transformation of the feature would be documented through traceability links.

It is obvious from the above that the addition of a feature in FArM does not dramatically change the PL architectural structure. This is because of the minimization of feature tangling. The new feature is either directly resolved or merged with existing features as a whole or partially. The merge elementary transformation does not increase feature tangling, since FArM does not allow for a merging transformation when the pre-existing feature is drastically changed. On the contrary, a merge is only allowed, when the pre-existing feature can be naturally extended to implement another feature or parts of its specification.

It is thus shown that the evolution of a FArM PL only causes the addition of new features or the natural extension of existing features, without significantly influencing the existing PL architectural structure. This enables in turn the respective extension of a configuration tool for the generative instantiation of products. For instance, the new feature along with its traceability links and dependencies is added to the tool configuration. This process requires significantly less effort than in the case of a PL architectural evolution that may cause the restructuring of the PL architecture.

Possible Products

Yet another problem of the generator-based model is the number of possible products that can be generated. Because of the use of a configuration tool or DSL, the possible component combinations are limited to the ones that have been already foreseen by the designer of the tool or DSL.

FArM allows for flexibility in the selection of the variability mechanisms used for product instantiation (sect. 5.3). That is, the PL developers are free to decide between a number of variability mechanisms and efficiently apply them on the feature level. Because a product can be defined as a set of features, any of these variability mechanisms can be utilized for product instantiation. For instance, the PL developers may efficiently use code-fragment superimposition for the generation of products. They may also utilize the plug in mechanisms and versatility of the Microkernel architectural style for product generation. In any case, the number of possible products in FArM is only limited by the number of possible feature combinations allowed due to the requires and excludes relations of the FM. These can be up to thousands of feature combinations [Boe02] and respective possible products depending on the number of features of the PL.

Inefficient Mapping

The generator-based model is based on the granularity of the extension components for the efficient instantiation of products. In the case where a feature is scattered and tangled into numerous extension components, it becomes very difficult to achieve the proper component granularity, so as to include exactly the selected features into a product. This leads very frequently to unwanted functionality in the final product, which reduces the product performance and increases its price. In cases of extreme feature scattering and tangling, it may also be impossible to perform a mapping of a feature to extension components.

FArM prevents such problems through the stronger feature-architecture mapping. The FArM components are derived directly from the features of the transformed FM, which are in turn directly traceable to the features of the initial FM. Thus, the selection of a feature in the initial FM always leads to a suitable set of architectural components. The inclusion of these components into a product leads also to the inclusion of the feature functionality. Furthermore, because of the minimization of feature tangling in FArM, there is little chance that the final product will receive unwanted features through the inclusion of a feature's components. The only features additionally included upon the inclusion of a feature are the ones defined by the requires and interacts relations of the feature.

5.5 Limitations

The FArM method has been designed with focus on maintainability and flexibility. Performance issues have also been taken into consideration in FArM. Furthermore, the method has been applied successfully to a variety of domains (chap. 3). This has allowed for the verification of the aforementioned FArM attributes. Nonetheless, there exist two issues that must be considered before FArM is selected for the development of a PL. These are performance and PL size.

With respect to performance, it is very likely that FArM cannot be efficiently applied for hard real-time systems. In time-critical situations, where responsiveness is of great importance, it is most likely that FArM architectures may not provide the required performance. This is due to the natural compromise between flexibility and maintainability against high performance. This fact does not totally exclude FArM for such domains, rather it relativises the advantages gained by the FArM method. For instance, several encapsulation and decoupling FArM practices would have to be ignored for hard real-time domains. Additionally, the conceptual integrity of the architecture would have to be jeopardized through the inclusion, e.g. of architectural entities that would explicitly boost performance, but would be unsuitable as features.

The next issue that has to be considered before the use of the FArM method, is the size of the PL that is to be developed. This must exceed a certain level, in order for the benefits of FArM to surpass the effort needed for its application. This level can be measured in terms of features in relation to their complexity and size. Empirically, it has been shown that the effort for the FArM application can be compensated for most industrial PLs. Nevertheless, FArM can also be used for small-sized PLs in the case where a reasonable number of product variations is to instantiated. This is because of the versatility that FArM allows through an efficient generative approach for product instantiation.

Chapter 6

Conclusions

At this point, the discussion of the various aspects of this work has been completed. Initially, a general state of the art analysis of the different PL methodologies was made, with respect to feature-architecture mapping, feature-level variability and product instantiation. Following this, special focus was placed on a few mature and representative state of the art PL methods. The identified problems of these methodologies provided the basis for a solution. This came in the form of the new Feature-Architecture Mapping (FArM) method. The contributions of this work, as well as the prospects for the future are presented in the upcoming sections.

6.1 Contributions

This work made a number of contributions during the development of the FArM method for the resolution of the state of the art open issues. A significant contribution is the overall and internal design of the FArM method itself for the enhancement of feature-architecture mapping. FArM defines three iterative transformation phases and a number of processes within each one. Namely, transformations based on Quality and NAR features, Architectural Requirements and Feature Interaction. The order of the transformation phases and the processes within them is especially designed to enable the methodical derivation of architectural components based on features and thus, to promote the enhancement of feature-architecture mapping. The first transformation phase assures the existence of only functional features in the transformed FM. This is a vital precondition for the implementation of the features' specification in architectural components. The second transformation phase provides a balance between the customer and architecture perspectives. This leads to a balanced mix of features originating from the problem and solution domains. The third FArM transformation phase optimizes the feature interaction, thus allowing for the eventual decoupling and encapsulation of the derived components. This leads to the enhancement of the system's maintainability. Throughout the application of the FArM transformation phases, the development of the PL architecture takes place. The Architecture Development phase of FArM assures that the PL architecture is gradually developed through the iterative specification of components.

Numerous contributions of this work can also be found within each of the FArM phases. In the first transformation phase, FArM provides the resolution of quality features on the feature level. Work has been done on the resolution of quality features on the architectural level [Bos00]. FArM makes use of techniques from such works, e.g. the profiles method and applies it on the feature level. This approach allows for the natural extension of pre-existing functional features and thus, for the indirect integration of quality attributes into the PL architecture from the feature level. Furthermore, FArM introduces the concept of Non-Architecture-Related (NAR) features. These features have a minimal impact on the software architecture and cannot be directly implemented in an architectural component. This fact deteriorates the mapping between features and the architecture. FArM methodically categorizes and resolves NAR features (sect. 4.3.1). These contributions allow for the enhancement of feature-architecture mapping.

Another contribution of FArM can be found in the second transformation phase, where architectural requirements are taken into consideration. As in the case of quality features, FArM handles architectural requirements on the feature level. More precisely, the requirements are integrated into existing functional features or new features are created for the satisfaction of the architectural requirements. This procedure minimizes the addition of architectural entities that may jeopardize the system's conceptual integrity, while at the same time it provides a balanced mix between the solution and problem domains. The latter is a vital precondition for the development of an efficient PL architecture.

In the third transformation phase, FArM performs transformations on features based on the feature interaction. For this purpose the concept of *feature interaction* was extended in FArM. FArM went beyond the classical definition of an interacts relation, by defining the so called *uses interacts relation* between features. This kind of interacts relation gives features knowledge of each other and enables the optimization of feature interaction and the direct derivation of component interfaces.

The optimization of feature interaction is done in FArM, among others, through the derivation or extension of hierarchy relations. To enable this optimization, FArM enhances the definition of FM hierarchy relations by enforcing aggregation and specialization. This approach allows for the localization of change on the architectural level. Components derived from FArM super-features can be used as encapsulation layers for the components derived by their sub-features. Because of the redirection of uses interacts relations to super-features and their integration into hierarchy relations, changes to components do not easily propagate to other components. This enhances the system maintainability. Additionally, due to the optimization of feature interaction, super-feature derived components can be utilized as switch mechanisms for sub-feature derived components. This is done without the addition of extra entities, thus allowing for the efficient application of variability mechanisms on the feature level.

The direct derivation of component interfaces from feature interactions is also another contribution of this work. This is achieved through the *uses* interacts relations between features of the transformed FM. Namely, the nature of the *uses* interacts relations allows for the direct derivation of requires and provides component interfaces. Additionally, this approach to component interface definition further improves the feature-architecture mapping.

Another important contribution of FArM is the derivation of architectural components from the features of the final transformed FM and their mapping to an architectural style. After the first FArM transformation phase, an architectural component is derived for each of the features of the transformed FM. The component specifications are gradually enhanced throughout the following transformation phases. Eventually, the components receive an interface based on the *uses* interacts relations of the respective feature. This process enables the direct derivation of architectural components based on features. The resulting architectural components are then placed in an architectural context. FArM has provided a mapping of the feature-derived components to various architectures, e.g. Microkernel, Layers, etc. (sect. 4.6).

Through the development of the FArM method, the goals of this work were achieved. Consequently, this work contributes to the enhancement of the mapping between features and the architecture, to the efficient variability on the feature level and to an improved generative product instantiation, as illustrated in the evaluation of this work (chapt. 5). The limitations of FArM were also identified with respect to the performance and the size of the PLs that can be developed with the method (sect. 5.5). Taking into consideration the limitations of FArM, in relation to the achievement of the aforementioned goals, FArM provides higher component stability, a clearer separation of concerns, as well as the preservation of the system's conceptual integrity. The latter contribute to enhanced maintainability, evolvability and a shorter time-to-market. All these attributes play a central role in the successful development of software PLs.

6.2 Future Work

The future of this work can be seen in three areas. In generic tool support, further domain applications and refactoring. In section 4.7, the various tools needed for the application of FArM were listed. It was also pointed out that an improvement of the FArM useability would be achieved through the development of a tool to generically support the FArM processes, e.g. transformation phases, traceability links, interaction optimization. The development of such a tool has been examined in a student-work [Kau05]. The results of this work denoted the possibility of the implementation of this tools as an Eclipse [Fou06] plugin. The development of such a tool should be a part of the the future work on FArM.

Another prospect for FArM would be its application in other domains for further refinement of its processes or its adaptation to domain classes. As already mentioned in section 3, FArM has been applied in a number of domains. Further application of FArM in other domains, e.g. soft real-time systems, the financial domain or even hard real-time systems, would enable the optimization of the FArM processes for these specific domains. Although key aspects of FArM have been explicitly quantitatively validated within the context of the NN-Trainer case study (sect. 5.2), it would be advantageous to perform an empirical quantitative evaluation of the FArM process and results in each of the aforementioned domains. This would include the rigorous definition of quantitative metrics, e.g. feature scattering and tangling as used for the NN-Trainer case study, for a variaty of usage scenarios typical to the domain at hand. This could be performed in the form of controlled experiments for each usage scenario. The optimization of the FArM processes could then lead to the adaptation of the FArM method for specific classes of domains, e.g. to the development of Real-Time FArM.

Finally, an important future prospect of the method is the refactoring of existing systems to comply to the FArM architecture model. Such future work would enable legacy and contemporary systems to take advantage of the FArM possibilities regarding maintainability and flexibility. This could be achieved, e.g. through the utilization of existing refactoring techniques and their combination with FArM.

Appendix A

NN-Trainer Feature Models

Figures A.1 and A.2 show the initial feature model and final transformed feature model respectively for the NN-Trainer case study.

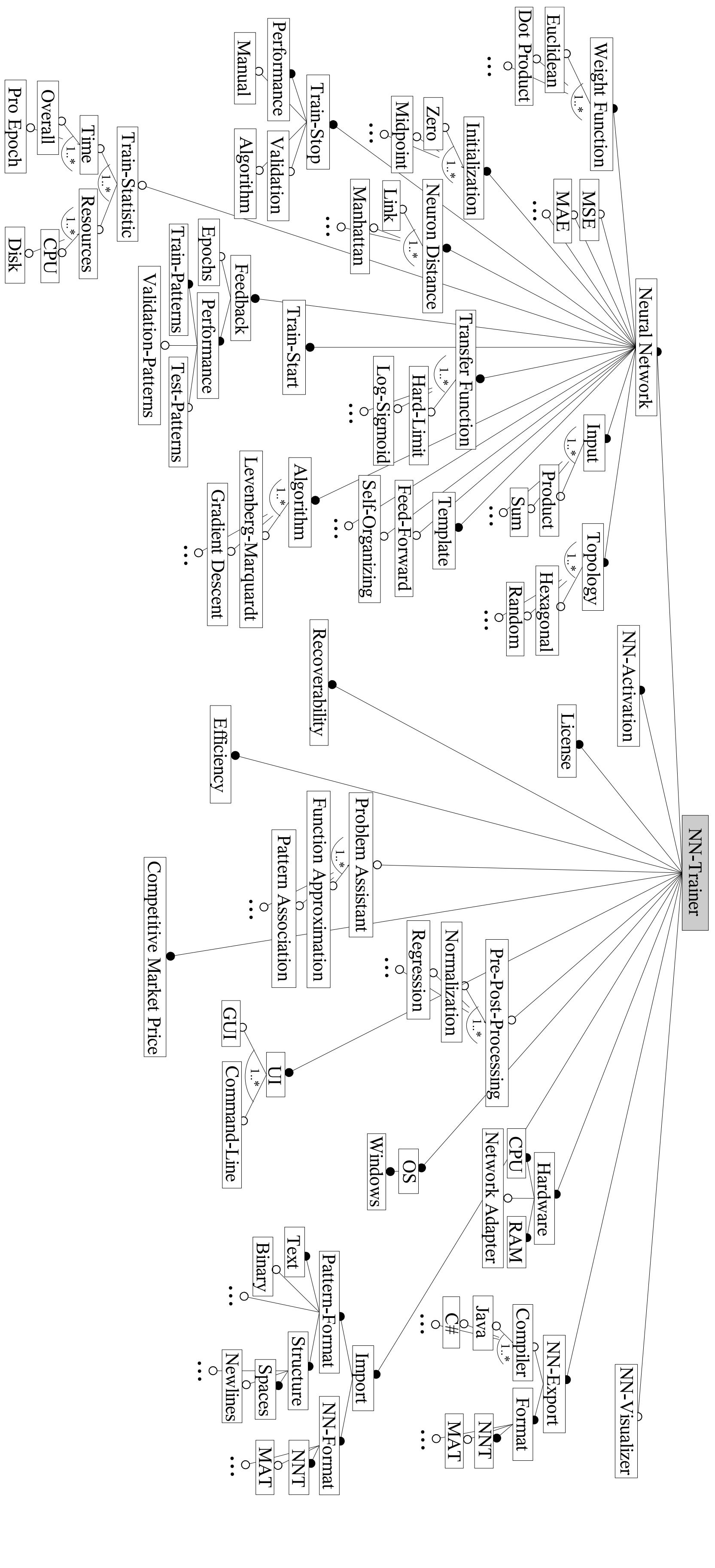


Figure A.1: NN-Trainer Initial FM

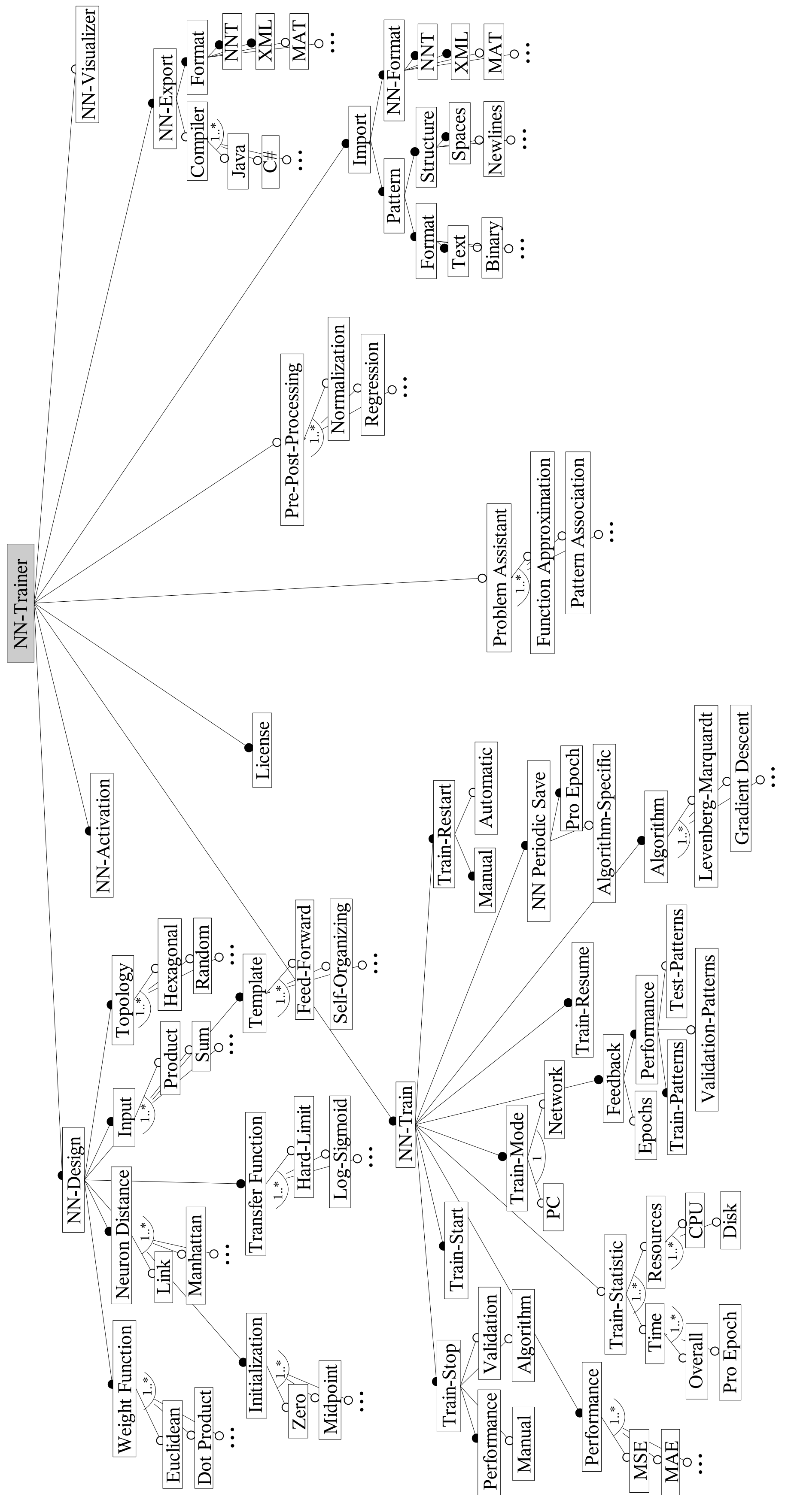


Figure A.2: NN-Trainer Final FM

Appendix B

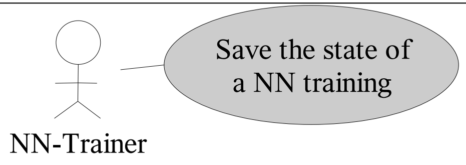
NN-Trainer Features Specification

This appendix provides the specification of a representative collection of features from the NN-Trainer case study. The specification of the NN-Trainer features has been performed with the FODA [PS94] domain analysis method. Additionally to the standard FODA feature specification form, a number of the use-cases related to the features are given.

Recoverability

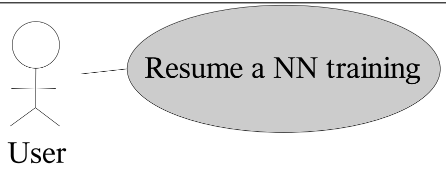
Name	**Recoverabiliy**
Synonyms	-
Description	The **Recoverability** feature allows the periodical saving of the state of a NN during its training. It allows the user to determine this period upon the initialization of the training. The training can then be resumed from the saved state.
Consists of	-
Source	NN-Trainer Requirements Specification v1.1
Type	-
Mutual Exclusive With	-
Mandatory With	**Train Start**, **NN Periodic Save**, **Resume**

Table B.1: **Recoverabiliy** feature FODA definition



Use Case Name	Save the state of a NN training
Preconditions	- A NN is being trained
Triggers	- A specified period of time elapses
Basic course of events	1. The NN training is paused 2. The NN structure is saved 3. The state of the training is saved 4. The NN training continues
Postconditions	- The last saved state of the NN training may now be retrieved

Table B.2: Save the state of a NN training use-case



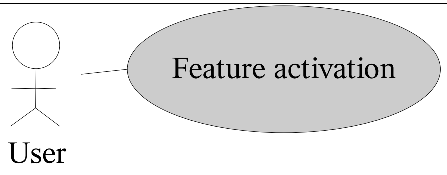
Use Case Name	Resume a NN training
Preconditions	- A NN has been irregularly terminated
Triggers	- The user resumes the training
Basic course of events	1. The NN structure is loaded 2. The NN training state is loaded 3. The user starts the training process
Postconditions	- The irregularly terminated training has been resumed from the last saved state

Table B.3: Resume a NN training use-case

Licence

Name	**Licence**
Synonyms	Licence Manager
Description	The **Licence** feature imposes the licensing policy of the NN-Trainer system. It allows the use of purchased functionality. Licensing can take place in multiuser, network environments. Any external licence manager software can be used for the imposement of the NN-Trainer licensing policy.
Consists of	-
Source	NN-Trainer Requirements Specification v1.1
Type	load-time
Mutual Exclusive With	-
Mandatory With	-

Table B.4: **Licence** feature FODA definition



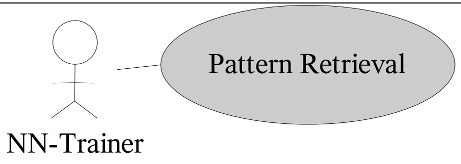
Use Case Name	Feature activation
Preconditions	- The feature is inactive
Triggers	- The user attempts to access the feature functionality
Basic course of events	1. A notification of an access attempt is made 2. The user is authenticated 3. The functionality access is approved 4. The user receives access to the functionality
Postconditions	- The feature is activated and the user can access the feature functionality

Table B.5: Feature activation use-case

Pattern

Name	**Pattern**
Synonyms	-
Description	The **Pattern** feature allows the loading of training and validation patterns. The format and structure of the patterns is read automatically. Numerous pattern formats and structures are supported by the feature.
Consists of	**Format**, **Structure**
Source	NN-Trainer Requirements Specification v1.1
Type	compile-time
Mutual Exclusive With	-
Mandatory With	**Train-Start**

Table B.6: **Pattern** feature FODA definition



Use Case Name	Pattern retrieval
Preconditions	- The pattern has already been imported
Triggers	- The training process requires access to the training or validation patterns
Basic course of events	1. A request for the retrieval of a specific pattern is made 2. The required pattern is retrieved 3. Access is provided to the specified pattern
Postconditions	- The pattern may be used by the caller

Table B.7: Pattern retrieval use-case

Appendix C

NN-Trainer Software Architecture

Figure C.1 shows a partial view of the software architecture for the NN-Trainer PL. The NN-Trainer software architecture is based on the Microkernel architectural style.

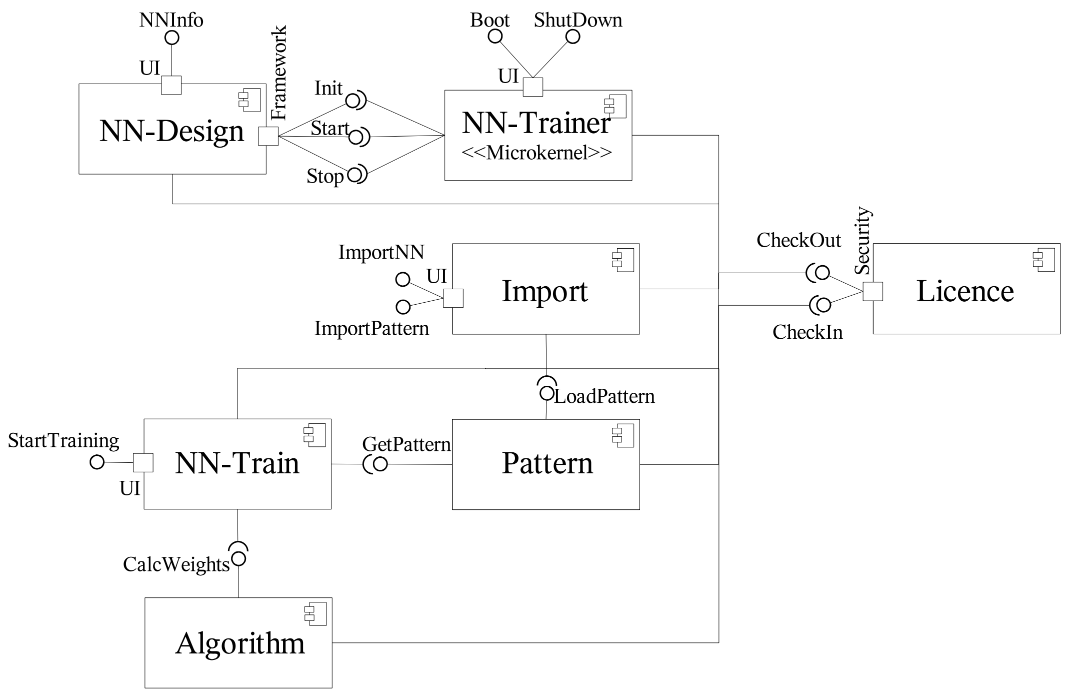


Figure C.1: A partial UML component diagram of the NN-Trainer software architecture

Bibliography

[A$^+$02] C. Atkinson et al. *Component-based Product Line Engineering with UML*. Addison-Wesley, 2002.

[B$^+$99] J. Bayer et al. PuLSE: A Methodology to Develop Software Product Lines. *In Proceedings of the 5th Symposium on Software Reusability (SSR '99)*, pages 122–131, 1999.

[BB01] Felix Bachmann and Len Bass. Managing variability in software architectures. *Proceedings of the 2001 symposium on Software reusability: putting software reuse in context*, pages 126–132, 2001.

[BG97] D. Batory and J.B. Geraci. Composition Validation and Subjectivity in GenVoca Generators. *IEEE Transactions on Software Engineering*, 23(2):67–82, 1997.

[BJM$^+$95] F. Buschmann, C. Jaekel, R. Meunier, H. Rohnert, and M. Stahl. *Pattern-Oriented Software Architecture-A System of Patterns*. John Wiley Sons, 1995.

[Boe02] K. Boellert. *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, Germany, 2002.

[Bor06] Borland. Together (R), Accessed on: 30.04.2006. Available from: `http://www.borland.com/de/products/together/index.html`.

[Bos00] J. Bosch. *Design & Use of Software Architectures - Adopting and Evolving a Product Line Approach*. Addison-Wesley, 2000.

[Bro95] F.P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Longman, 1995.

[Bus96] F. Buschmann. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

[CKM$^+$03] M. Calder, M. Kolberg, M.H. Magill, et al. Feature Interaction - A Critical Review and Considered Forecast. *Elsevier: Computer Networks*, 41(1):115–141, 2003.

[CN01] P. Clements and L. M. Northrop. *Software Product Lines: Practices and Patterns*. Addison Wesley Professional, 1st edition, Aug 20 2001.

[Fou06] The Eclipse Foundation. Eclipse, Accessed on: 30.04.2006. Available from: `http://www.eclipse.org/`.

[GAd98] D. Griss, R. Allen, and M. d'Allesandro. Integrating Feature Modelling with the RSEB. In *5th International Conference of Software Reuse (ICSR-5)*, 1998.

[GBS01] Jilles Van Gurp, Jan Bosch, and Mikael Svahnberg. On the Notion of Variability in Software Product Lines. *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, 00:45–54, 2001.

[Gib97] J. Gibson. Feature Requirements Models: Understanding Interactions. *Dini et al.*, pages 46–60, 1997.

[Hat06] Red Hat. RPM Package Manager, Accessed on: 30.04.2006. Available from: `https://www.redhat.com/`.

[HDB96] M. T. Hagan, H. B. Demuth, and M. H. Beale. *Neural Network Design*. PWS Publishing, 1996.

[Hon06] Honeywell. DOME (DOmain Modelling Environment), Accessed on: 6.05.2006. Available from: `http://www.htc.honeywell.com/dome/`.

[IBM06] IBM. Rational RequisitePro, Accessed on: 30.04.2006. Available from: `http://www-306.ibm.com/software/awdtools/reqpro/`.

[Inn06] Foerderung von Innovativen Netzwerken - Magentisches Monitoring (MagMon) / Promotion of Innovative Networks - Magnetic Monitoring. InnoNet, Accessed on: 22.02.2006. Available from: `http://www.vdivde-it.de/innonet/projekte/in_pp057_magmon.pdf`.

[ISO01] (ISO) International Standardization Organization. ISO/IEC 9126-1:2001, Software Engineering, Product Quality, Part 1: Quality Model, 2001.

[JF98] R. Johnson and B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2), 22-5 1998.

[JGJ97] I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success.* Addison-Wesley, 1997.

[JZ05] W. Jirapanthong and A. Zisman. Supporting Product Line Development through Traceability. *12th Asia-Pacific Software Engineering Conference (APSEC'05)*, pages 506–514, 2005.

[Kau05] Michel Kaufmann. Analyse und Evaluation der Modellierungsaspekte der Methode "Feature-Architecture Mapping" Fallstudie: BlackBerry Produktlinie. TU-Ilmenau, 10 2005.

[KCH$^+$90] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, 1990.

[Kic97] G. Kiczales. Aspect-Oriented Programming. In *European Conference on Object-Oriented Programming (ECOOP '97)*, pages 220–242. Springer-Verlag, 1997.

[Kic01] G. Kiczales. Getting Started with AspectJ. *Communications of the ACM*, 44(10):59–65, 2001.

[KLD02] K. C. Kang, J. Lee, and P. Donohoe. Feature-Oriented Product Line Engineering. *IEEE Software*, 9(4):58–65, Jul./Aug. 2002.

[Ltd06] Techsoft Pvt. Ltd. Matrix TCL Pro - Matrix Algebra Made Easy. TechSoft, Accessed on: 08.04.2006. Available from: `http://www.techsoftpl.com/matrix/index.htm`.

[Mac05] Macrovision. *End Users Guide.* Macrovision Corporation, 9.5 edition, August 2005.

[Mei06] J. Meister. *Product-Driver Development of Software Product Lines applied on an Example of Analytical Software (Produktgetriebene Entwicklung von Software-Produktlinien am Beispiel analytischer Anwendungssoftware).* PhD thesis, University of Oldenburg, 2006.

[Mic06] Microsoft. Microsoft Visual Studio, Accessed on: 30.04.2006. Available from: `http://www.microsoft.com/germany/msdn/vstools/default.mspx`.

[MR02] K. Mohan and B. Ramesh. Managing Variability with Traceability in Product and Service Families. *35th Annual Hawaii International Conference on System Sciences (HICSS'02)*, 3:76, 2002.

[oS06] University of Stuttgart. Stuttgart Neural Network Simulator (SNNS), Accessed on: 21.07.2006. Available from: `http://www-ra.informatik.uni-tuebingen.de/SNNS/`.

[OT01] H. Ossher and P. Tarr. *Software Architectures and Component Technology*, chapter Multi-Dimensional Separation of Concerns and the Hyperspace Approach. Kluwer Academic Publishers, 2001.

[Ous98] J.K. Ousterhout. Scripting: Higher Level Programming for the 21st Century. *IEEE Computer*, May 1998.

[Par72] D. L. Parnas. On the criteria to be use in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[PS94] A. S. Peterson and J. L. Jr. Stanley. Mapping a domain model and architecture to a generic design. Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213, May 1994.

[PS06] PnP-Software. XFeature – Feature Modelling Tool, Accessed on: 30.04.2006. Available from: `http://www.pnp-software.com/XFeature/`.

[Rie03] M. Riebisch. Towards a More Precise Definition of Feature Models. In *Workshop at ECOOP*, pages 64–76. BookOnDemand Publ. Co., 2003.

[SEI05] (SEI) Software Engineering Institute. Software Product Lines. Carnegie Mellon University, Accessed on: 13.07.2005. Available from: `http://www.sei.cmu.edu/productlines/index.html`.

[Soc04] P. Sochos. Mapping Feature Models to the Architecture. *Proceedings of the First International Software Product Lines Young Researchers Workshop (SPLYR)*, pages 51–60, 2004.

[SPC93] (SPC) Software Productivity Consortium. Reuse-driven software processes guidebook. Technical Report SPC-92019-CMC, Version 02.00.03, Herndon, VA: Software Productivity Consortium, 1993.

[SRP04] P. Sochos, M. Riebisch, and I. Philippow. Feature-Oriented Development of Software Product Lines: Mapping Feature Models to the Architecture. *Proceedings of Net.ObjectDays*, pages 138–152, 2004.

[SRP05] P. Sochos, M. Riebisch, and I. Philippow. Feature-Oriented Architecture Design for Maintainability and Evolution of Product Lines. *Proceeding of Software Engineering*, March 2005. (in german).

[SRP06] P. Sochos, M. Riebisch, and I. Philippow. The Feature-Architecture Mapping (FArM) Method for Feature-Oriented Development of Software Product Lines. *Proceeding of Computer Based Systems Engineering (ECBS 06)*, March 2006.

[Str03] D. Streitferdt. *Family-Oriented Requirements Engineering.* PhD thesis, Faculty of Informatics and Automation of the University of Ilmenau, August 2003. (in german).

[SY99] J. Suzuki and Y. Yamamoto. Extending UML with Aspects: Aspect Support in the Design Phase. In *Aspect-Oriented Programming Workshop at ECOOP '99*, 1999.

[Tar05] Peri Tarr. Multi-Dimensional Separation of Concerns: Software Engineering using Hyperspaces. IBM T. J. Watson Research Center, Accessed on: 9.11.2005. Available from: `http://www.research.ibm.com/hyperspace/`.

[TM06] Inc. The MathWorks. Neural Network Toolbox, Accessed on: 22.07.2006. Available from: `http://www.mathworks.com/products/neuralnet/`.

[TO01] P. Tarr and H. Ossher. *Hyper/J User and Installation Manual*, 2001.

[WL99] D. M. Weiss and C. T. R. Lai. *Software product-line engineering : a family-based software development process.* Addison-Wesley, 1999.

Theses

1. Separation of concerns is a vital precondition for advanced maintainability and high flexibility of software product lines.

2. Features pose the main concerns in the context of software product lines.

3. Ideally, one feature should be implemented into exactly one architectural component. In practise, a strong mapping between features and the architecture must exist.

4. The Feature-Architecture Mapping (FArM) method developed in this work, allows for a stronger mapping between features and the architecture in comparison to the contemporary state of the art product line development methodologies and other approaches.

5. FArM progressively transforms an initial customer-specific feature model producing a final transformed feature model, where the application logic of each feature can be implemented into exactly one architectural component. The feature interaction is reflected by the component communication.

6. Throughout the FArM transformation phases optimization of feature interaction is performed with focus on system maintainability and variability.

7. The stronger feature-architecture mapping achieved through the FArM processes allows for an efficient variability on the feature level and for an improved generative product instantiation compared to the present state of the art approaches.

Ilmenau, April 22, 2007

Erklärung

Ich versichere, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Weitere Personen waren an der inhaltlich materiellen Erstellung der vorliegenden Arbeit nicht beteiligt. Insbesondere habe ich hierfür nicht die entgeltliche Hilfe von Vermittlungs- bzw. Beratungsdiensten (Promotionsberater oder anderer Personen) in Anspruch genommen. Niemand hat von mir unmittelbar oder mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen. Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form einer Prüfungsbehörde vorgelegt. Ich bin darauf hingewiesen worden, dass die Unrichtigkeit der vorstehenden Erklärung als Täuschungsversuch angesehen wird und den erfolglosen Abbruch des Promotionsverfahrens zur Folge hat.

Ilmenau, April 22, 2007

Periklis Sochos