Otto-von-Guericke-Universität Magdeburg

Fakultät für Informatik
Institut für Technische und Betriebliche Informationssysteme

Master Thesis

Modeling and Implementation of Dependent Software Product Lines

Author:

Tao Wei

2nd June 2009

Advisor:

Prof. Dr. rer. nat. habil. Gunter Saake,
Dipl.-Inform. Marko Rosenmüller,
Dipl.-Inform. Norbert Siegmund

Universität Magdeburg
Fakultät für Informatik
Postfach 4120, D–39016 Magdeburg
Germany

Wei, Tao:
Modeling and Implementation of Dependent Software Product Lines
Masterarbeit Otto-von-Guericke-Universität Magdeburg, 2009.

Acknowledgements

I would like to thank my master thesis advisors for their support and assistance. I would like to thank Prof. Dr. Gunter Saake for his support and confidence in me and allowing me to work on master thesis in his group Institut für Technische und Betriebliche Informationssysteme. This paper is completed under the guidance of Marko Rosenmüller and Norbert Siegmund. I would like to thank Marko Rosenmüller, Norbert Siegmund and Martin Kuhlemann for their helpful reviews and suggestions. I would like especially thank Marko Rosenmüller and Norbert Siegmund, they were specifically assigned to me for the thesis and without our lengthy discussions about various related topics, this thesis would not have been possible. Marko Rosenmüller gave me many suggestions on the topic of dependent Software Product Lines and tried to answer my innumerable questions orally or through emails. I would like also to thank my friend Luo Lei, Cao Shuai, and Syed Saif ur Rahman who gave me some good suggestions for this paper. Finally I would like to thank my family because of whom this education is made possible for me, and my girlfriend Ying, when I was in need of help, she always encourage me.

Contents

Contents ii

List of Figures iv

List of Tables vi

List of Abbreviations vii

1 Introduction **1**

1.1 Motivation 1

1.2 Goals 3

1.3 Structure of the Thesis 4

2 Background **5**

2.1 Software Product Line Engineering 5

2.1.1 Feature-Oriented Domain Analysis 8

2.1.2 Staged Configuration Using Feature Models 10

2.1.3 Feature-Oriented Programing 12

2.2 Dependent Software Product Lines 15

2.3 Sensor Networks 18

2.4 Summary 21

3 Modeling Dependent Software Product Lines **23**

3.1 Case Study: Sensor Network Product Line 23

3.1.1 Product Lines for a Sensor Network 24

3.1.2 Integrating Multiple Product Lines 30

3.2 Using FM-References for Modeling Dependent Product Lines 31

3.3 Modeling Software Product Line Instances 34
3.4 Discussion 38

4 Implementation 39

4.1 Sensor Networks 39
4.1.1 Dependent Software Product Lines 40
4.1.2 Implementation of Communication Product Line 41
4.1.3 Implementation of Dependent Software Product Lines 43
4.2 Configuring Dependent Software Product Lines 48
4.3 Summary 49

5 Evaluation 50

5.1 Evaluation of Modeling Approaches 50
5.2 Evaluation of Implementation Approaches 53

6 Conclusion 55

6.1 Further Work 56

Bibliography 58

Appendix 61

List of Figures

1.1 Sensor Network 2

2.1 Software development based on Domain Engineering [CE00] 6
2.2 Structure of the SEI Framework for Product Line Practice [CE00] 7
2.3 Types of feature realtions 8
2.4 Example of feature diagram 9
2.5 Cardinality-Based feature diagram 10
2.6 Staged Configuration by eliminating features 11
2.7 Unfolding a feature diagram reference [CHE04] 12
2.8 Implementing refinements by mixin inheritance [BSR04] 14
2.9 Expression and refinement chains 15
2.10 A Sensor Network SPL using a Communication SPL and Client Application SPL 16
2.11 Dependent SPLs 17
2.12 A Sensor Network SPL using different instances of Communication SPL and Client Application SPL 17
2.13 Sensor Network Node Architecture [SMZ07] 19
2.14 Three types of messages of SPIN routing protocol 20

3.1 Feature Diagram for Sensor Network Nodes 24
3.2 Feature Diagram for Client Applications 26
3.3 Feature Diagram for Communication 28
3.4 Feature diagram for FAME-DBMS [RSS$^+$08] 29
3.5 SPL for a Sensor Network 30
3.6 Dependent Software Product Lines 31

3.7 Using FM-References to model Dependencies between SensorNetwork Product Line and the underlying SensorNetworkNode, FAME-DBMS, and Communication Product Lines, adopt from [CHE04] 32
3.8 Using FM-References to model Dependencies between SensorNetwork Product Line and the underlying ClientApplication and Communication Product Lines, adopt from [CHE04] 33
3.9 Dependencies between SPLs and instances [RSKuR08] 35

4.1 Dependencies between different Product Lines 41
4.2 Class Diagrams of Communication Product Line 42
4.3 Classes and Refinements (Layers) 43
4.4 The Equation File for Communication Product Line 43
4.5 Unfolding Feature Model References 45
4.6 Staged Configuration of Communication Product Line by removing Feature Compression 46

List of Tables

3.1 Domain Constraints 37
3.2 Instance Constraints 37
3.3 Conditional Dependencies 38

5.1 Comparison of Feature Model References and Modeling SPL Instances . . 52

List of Abbreviations

FODA	Feature-Oriented Domain Analysis
FOP	Feature-Oriented Programing
OOP	Object-Oriented Programing
SPL	Software Product Line
SPLE	Software Product Line Engineering
CM	Configuration Management
DSL	Domain Specific Languages
PDA	Personal Digital Assistants
UML	Unified Modeling Language
AHEAD	Algebraic Hierarchical Equations for Application Design
FC++	Feature-Oriented C++
SIG	Special Interest Group
Wi-Fi	Wireless Fidelity
SPIN	Sensor Protocol for Information via Negotiation
MCIC	Multi-node Cooperative Image Compression

TCP/IP	Transmission Control Protocol/Internet Protocol
UDP	User Datagram Protocol
AES	Advanced Encryption Standard
DES	Data Encryption Standard
DSA	Digital Signature Algorithm
RSA	Rivest Shamir Adleman
ECC	Elliptic Curves Cryptography
LRU	Least Recently Used
LFU	Least Frequently Used
FOSA	Feature-Oriented Software Development
OCL	Object Constraint Language
PLM	Product Line Model

Chapter 1

Introduction

1.1 Motivation

Car Product Lines are used to produce cars which have different functionalities for different customers. For example, the transmission of a car can be manual or automatic. If a customer needs the functionality of manual, we can choose this functionality, if a customer needs the functionality of automatic, then we can choose this functionality. Similarly, Software Product Lines (SPLs) are used to develop software which can be customized to different use cases [CE00]. A member of SPL is built from a common set of reusable assets, and to satisfy special requirement of a customer. Software Product Line Engineering (SPLE) has to be explained in two parallel parts: Domain Engineering and Application Engineering. Domain Engineering consists of Domain Analysis, Domain Design and Domain Implementation [PBvdL05]. The first step of Domain Engineering is Domain Analysis, which means we will analyze all possible features and create a model of a specific domain or real system. The common and variable properties of product families can be recognized as features, e.g., transmission, manual and automatic are features of a car. The method of Feature-Oriented Domain Analysis (FODA) is explained by Czarnecki et al. in Generative Programming [CE00]. Feature modeling is proposed as part of the FODA and consists of several feature diagrams. Feature modeling is well suited for capturing the commonalities and variability in program families. New user requirements will be discovered in Application Engineering and we have to extend our feature diagrams according to the new user requirements.

As an explanation for the domain, we will use a case study sensor network scenario. Figure 1.1 depicts a simple sensor network which includes sensor nodes, PDA and Laptop. The type of sensor nodes can be divided into sensor node, data storage node and access node. In the Domain Analysis, sensor network node can be considered as a product line, which develops programs for different types of network nodes. The client application can also be considered as a product line, which develops programs for different hardware interfacing with sensor network nodes, e.g., Laptop or PDA. We can use different protocols to implement the communication between sensor nodes and client applications, e.g., Bluetooth, Wi-Fi, etc. That is why we need Domain Analysis, and use feature models to capture all possible commonalities and variability. With respect of assets reuse, communication has to be considered as a product line to provide different communication functionalities.

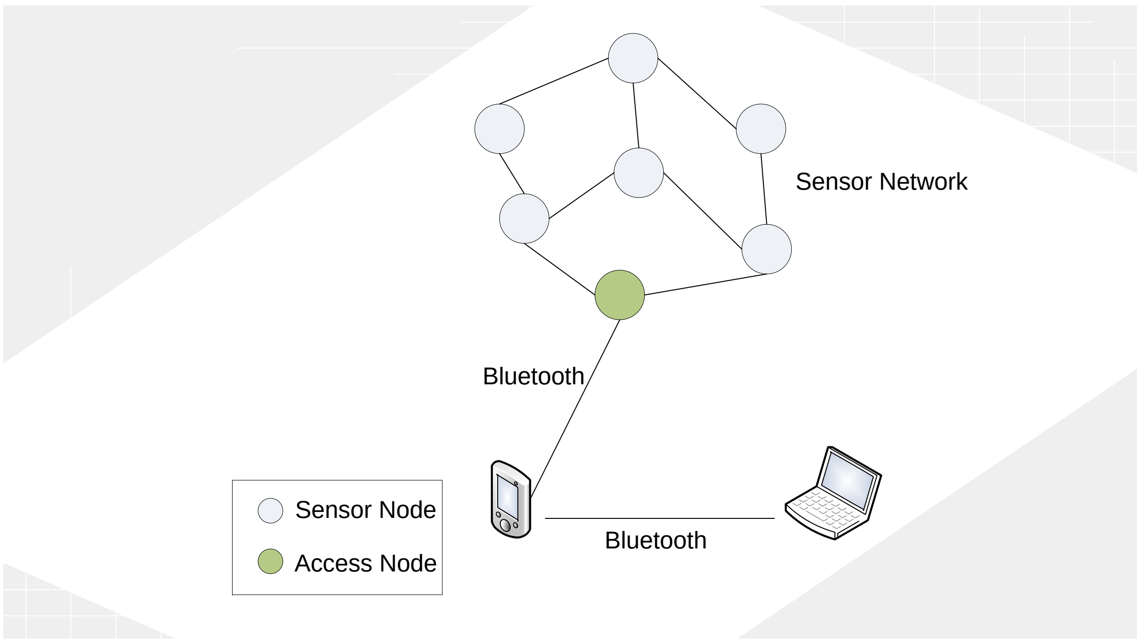


Figure 1.1: Sensor Network

We can derive a concrete software program using configuration. Variants are created in order to support different sets of requirements in program families. In the context of variant configuration, each product can be seen as an individual system variant. A feature-based configuration tool allows us to load the feature model of a program family and specify a system variant by selecting needed features from the feature model [CW04]. Sometimes multiple product lines are integrated into one product line, which needs functionalities or components from those involving SPLs. In this case, we can have imagination that between the constituent SPLs will have many dependencies and constraints. If we want to configure one SPL which needs functionalities provided by other product lines, we have to consider about constraints and dependencies between them. For example, if the encryption method between sensor node and a client application like PDA is DES, feature DES has to be selected in communication product line. So product lines for sensor network nodes and client applications may depend on the communication product line. That means, in such compositions of SPLs, all involved product lines may be dependent on each other. When creating products from these dependent product lines all created products (SPL instances) have to be compatible. So the configuration of each concrete SPL instance has to be defined [RSKuR08]. Furthermore, product lines may have dependencies and constraints with other SPL instances. We have to use the feature modeling and new approaches of modeling diagrams, e.g., staged configuration, to describe SPL instances and modeling dependencies between them.

Development of similar programs using SPLs provides a high degree of reuse. Sometimes, an SPL may need functionalities provided by other product lines, so there are constraints and dependencies between involved product lines. In Domain Analysis we have to find out the constraints and dependencies between dependent product lines. Feature diagrams are used to capture features in a problem domain (e.g., sensor network). Feature model references can be used to describe dependencies between different product lines. However, using feature model references we can only describe dependencies and constraints on a domain model level. An SPL may depend on a concrete product of other product line (SPL instance), so we will meet some problems when using feature model references to describe dependencies and constraints between them. In this thesis we will explore new approaches which are based on the feature diagrams to clearly describe the constraints and dependencies between dependent SPLs.

1.2 Goals

In this thesis, we first review the SPLE and FODA. In FODA, feature diagram can be used for modeling commonalities and variability of the problem domain. However, usually multiple small SPLs have to be integrated into one larger SPL. We will use a sensor network scenario to describe the dependencies and constraints between dependent SPLs. For example, an SPL for sensor node software should be integrated into a larger SPL for a whole sensor network. In order to provide the functionality of communication in sensor networks, a Communication product line has to be considered in this thesis, and all possible features of Communication product line will be analyzed. The Communication product line can be integrated into an SPL for sensor node software to fulfill the functionality of communication between sensor nodes. In order to avoid manual configuration of each of the smaller SPL instances, the configuration of the smaller SPLs should depend on the configuration of surrounding larger SPLs.

- Development and analysis of approaches for composition of multiple SPLs.
- Analysis of possible features of a communication SPL that is used for evaluation of the approach.
- Development of small SPLs to simulate or implement a sensor network for evaluation.

- Evaluation of different composition scenarios considering multiple instances of the same SPL.

We will use Staged Configuration and Stepwise Refinement to analyze the Sensor Network Scenario. We will find out the constraints between dependent SPLs and describe that using FM-References cannot clearly describe the constraints between dependent SPLs, and then we will present an extension of current feature diagrams to describe the dependencies and constraints between concrete SPL instances. We will describe dependencies between all SPLs and also between concrete SPL instances of sensor networks.

1.3 Structure of the Thesis

Chapter 2 This chapter is about the background of SPLE. We review the Feature-Oriented Domain Analysis (FODA), Feature-Oriented Programming (FOP), and Staged Configuration using Feature Modeling. This section is also about the background of Dependent Software Product Lines and Sensor Networks. We will review the communication protocol, e.g., Bluetooth.

Chapter 3 In this chapter we will bring a Sensor Network scenario, and modeling the feature diagrams for this scenario. The references and constraints between different product lines will be explained in this chapter.

Chapter 4 This chapter is focus on the Implementation of the sensor network scenario. We will use the Ehtenet to realize the communications between sensor network nodes and client applications. A real sensor network will be implemented for the further evaluation.

Chapter 5 This chapter is about evaluation which based on the existing FAME-DBMS SPL and additionally smaller SPLs. Dependencies between SPL instances have to be analyzed and represented in a model, and different composition scenarios have to be evaluated.

Chapter 6 we will present the conclusions of this thesis and discuss the further Work about the dependent Software Product Lines.

Chapter 2

Background

In this chapter we explore various topics required as background for understanding Software Product Line Engineering and dependent Software Product Lines. We review Feature-Oriented Domain Analysis (FODA) and Feature-Oriented Programming (FOP). We explore the staged configuration using feature models. Finally we provide foundations of Sensor Networks and communication protocols, e.g., Bluetooth.

2.1 Software Product Line Engineering

In [PBvdL05] Klaus Pohl et al. define the term software product line engineering like: “Software product line engineering is a paradigm to develop software applications (software-intensive systems and software products) using platforms and mass customization.” They have defined that the mass customization is the large-scale production of goods tailored to individual customers’ needs. This definition comes from a car product line, at which different customers may need different cars. The same as the car production, SPL is used for development of different software products. An SPL is a set of software systems, which share common features, and management of specific need to meet a particular task. Software Engineering Institute Carnegie Mellon University defines SPL as: “A software product line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying 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.”

Domain Engineering and Application Engineering are two parallel parts of SPLE. If we want to build a new system, we can use Domain Engineering to collect, organize, and store past experiences in developing of systems in a particular domain in form of reusable assets, e.g., in our sensor network scenario, communication product line is useful for the communication among sensor nodes, so we consider it as a SPL, and if we want to develop a concrete sensor network system, we can use it as reusable asset.

Domain Engineering has three parts: Domain Analysis, Domain Design, and Domain Implementation. Application Engineering reuses the results of Domain Engineering. Application Engineering is the process of producing concrete systems using the reusable assets developed during Domain Engineering [CE00]. We will use a figure to describe the relationship between Domain Engineering and Application Engineering.

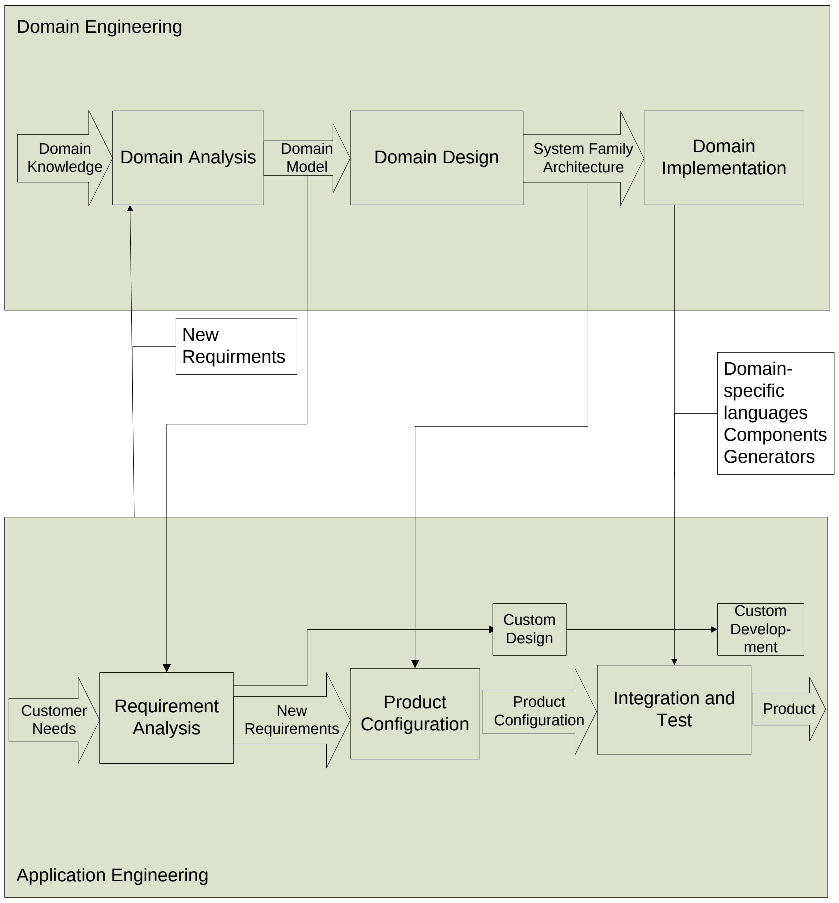


Figure 2.1: Software development based on Domain Engineering [CE00]

Domain Engineering and Application Engineering are two parallel processes in software development. Domain Analysis is the first step of Domain Engineering. Czarnecki et al. describe that Domain Analysis is a process, which analyze and create a model of a special domain [CE00]. We have to analyze all possible features in a special domain, e.g., sensor networks. Then we have to create a model for sensor network using feature diagrams. Generally one feature model may consist of multiple feature diagrams. One feature diagram represents an SPL. In our sensor network scenario, we also have more than one feature diagrams, e.g., communication SPL, sensor network SPL and so on. The domain model has to describe the common and variable properties of all systems of the domain, and describe the dependencies among these properties. In Domain Engineering all possibilities have to be considered, but customer may have new requirement for the system. In this case, we don't have the assets, if a customer needs a new functionality, then we have to take the new requirement to the Domain Engineering, and modify our feature diagrams. For example, in our sensor network, if a customer needs the functionality of encryption, and we don't have the feature of encryption in our feature diagrams, so we have to modify our feature diagrams in this case. Figure 2.1 shows relationships between Domain Engineering and Application Engineering.

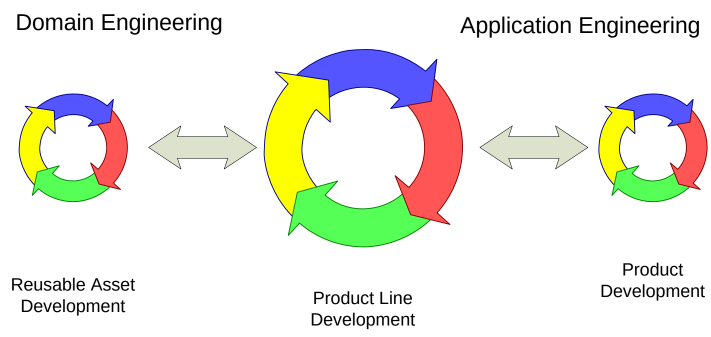


Figure 2.2: Structure of the SEI Framework for Product Line Practice [CE00]

After we have a review of the Domain Engineering, let us have a review of the Application Engineering, Application Engineering is the process of building systems, which based on the results of Domain Engineering. On the requirement analysis of a new kind of concrete application, we use existing domain model and customer requirements. We have to describe customer needs using features, which represent the reusable requirements from the domain model. Of course if the customer requirements are not found in the domain model, then it requires for the new custom development. The new requirements should also be fed back to Domain Engineering, we use this step to refine and extend the reusable assets [CE00]. Domain Engineering is used for reusable asset development, and Application Engineering is used for product development. The process of Domain Engineering and Application Engineering are useful for the SPL development, as shown in Figure 2.2.

In Domain Engineering, domain is as the real world and a set of systems. It encapsulates the knowledge of a problem area, such as in a bank accounting domain, it includes concepts of accounts, customers, withdraws and deposits. In the Unified Modeling Language (UML), domain means an area of knowledge or activity characterized by a set of concepts and terminology understood by practitioners in that area.

2.1.1 Feature-Oriented Domain Analysis

Feature-Oriented Domain Analysis (FODA) is a Domain Analysis method which focuses on the features of domain, e.g., sensor networks. We will explain FODA in two parts: definition of domain analysis which is the first step of Domain Engineering, and explanation of the feature-oriented part of FODA. In Domain Analysis we will identify and define the domain and the scope of the domain, then we will create the domain model which describes the common and variable properties of all systems of the domain, finally, we will define and find out the dependencies and constraints between these properties. We have made a brief review of Domain Analysis in the section of SPLE, Domain Analysis contains the finding of common and variable features of all systems, which belong to the domain. According to FODA, a feature means an aspect of a system, and is visible to the end-user of the systems of the domain. For example, transmission of a car is a feature of the domain, automatic and manual are two sub-features of feature transmission, which are visible to end-users [CE00].

In FODA, feature model consists of feature diagrams, feature definitions, composition rules and rationale for features. Feature diagram describes the decomposition of features into sub-features. For example, automatic and manual are sub-features of feature transmission. For each sub-feature it can be defined as mandatory, or, alternative, and optional. Feature definitions describe all features of all systems of the domain and their types. We have two types of composition rules at FODA: requires rules and mutually-exclusive-with rules [CE00]. The definition of requires rules is that one feature requires another feature as condition. The definition of mutually-exclusive-with rules is that one feature cannot exist if the other feature exists. For example, the feature of automatic is mutually exclusive with the feature of manual. Rationales for features describe the reason for choosing a feature or not. In a feature diagram, we can add the rationales for feature as annotations to the features [CE00].

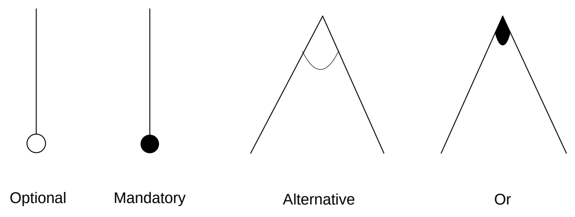


Figure 2.3: Types of feature realtions

Context analysis and domain modeling are two phases of analysis process at FODA. Context analysis defines the context of the domain, e.g., the scoping of all systems of the domain, requires between features, the constraints and dependencies. In the phase of domain modeling, a domain model should be produced. Domain modeling consists of information analysis, feature analysis and operational analysis. In the step of information analysis, we will create of a information model, which consists of the domain entities and the relationship between them. In the step of feature analysis, we will create a feature model, which will be described below. In the step of operational analysis, we will create an operational model which describes the data flow and control in the application domain and the relationships between the objects and the features of the feature model [CE00].

Feature Models Czarnecki et al. described feature models in Generative Programming [CE00]. Feature models will be used in the section of staged configuration, so we make a brief review of feature models in this section. Firstly, a feature diagram is a tree with the root and its descendent nodes. The root represents a concept, e.g., a concrete sensor network system. Its descendent nodes are features which belong to this software system. Features in FODA can be mandatory, optional, or, and alternative. As shown in Figure 2.3.

The root of a feature diagram represents a concept, so in Figure 2.4, we refer to C as the concept node. The remaining nodes represent features in feature diagram. A mandatory feature must be selected if its parent is selected in a software system. For example, if we want to implement the functionality of C, feature f1 must be selected, because it is mandatory. The optional feature means that, in the description of a concept, it may be included or not, it is not necessary, it depends on the use case. If the user needs the functionality, then select it. In this feature diagram, f2, f3 and f6 are optional features. Alternative features mean that, among all of the descendent features only one feature can be selected. For example, f1 is mandatory, so it must be selected, f4 and f5 are sub-features of feature f1, so one feature of f4 and f5 must be selected. Or features mean that, one or more than one features can be selected, if we want to configure an SPL or a software system. In Figure 2.4, f3 is an optional feature of C, f7, f8, f9 are sub-features of f3, so, if f3 is selected, one or more than one features of f7, f8 and f9 must be selected.

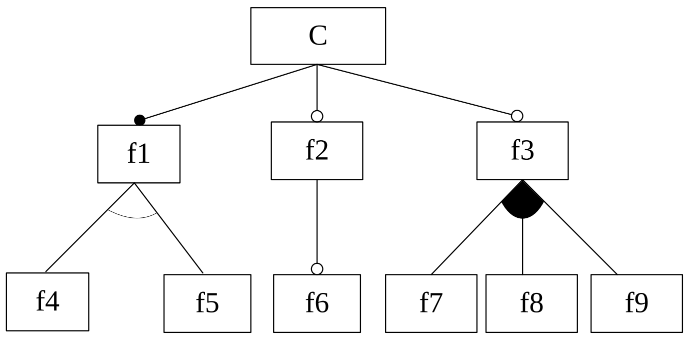


Figure 2.4: Example of feature diagram

Figure 2.5 shows the cardinality-based feature modeling, this feature diagram describes that at least two and at most four features have to be selected when we want to implement the functionality of the software system. The cardinality-based feature modeling is useful for staged configuration, which will be explained in the next section.

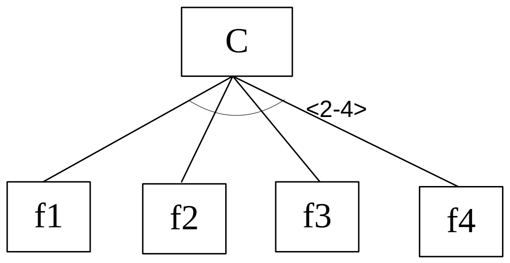


Figure 2.5: Cardinality-Based feature diagram

FODA divides all features in three types according to their binding time: compile-time features, activation-time features and runtime features. Compile-time features are bound to an application at compile time. Activation-time features are bound each time the application starts, but activation-time features are stable during execution time. Runtime features are bound dynamically during the execution of applications. The process of FODA consists of two phases: context analysis and domain modeling. The scoping of the domain and the constraints are defined in the context analysis. We have to produce a feature model in domain modeling after domain analysis [CE00].

2.1.2 Staged Configuration Using Feature Models

Generally, the relationship between feature models and configuration is just the same as the relationship between the class and the instance in the Object-Oriented programming (OOP) [CE00]. In this section we will describe staged configuration based on feature models. The process of specifying a family member can be performed in stages, where each stage eliminates some configuration choices, as shown in Figure 2.6. This process can be referred as staged configuration. Each stage takes a feature model, and produces a specialized feature model. We can obtain various systems by selecting the desired features from feature models. Then we will describe the relationship between specialization and configuration, specialization is useful if configuration needs to be performed in stages.

Czarnecki et al. define the definition of staged configuration in [CHE04] as: "The process of specifying a family member may also be performed in stages, where each stage eliminates some configuration choices. We refer to this process as staged configuration." Each stage configures a new feature diagram, and eliminates features which we don't need for the special software system. Then we also have to know the idea of specialization.

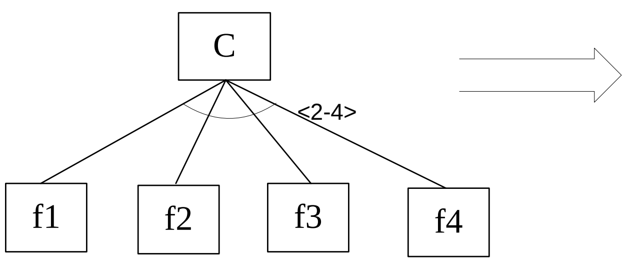


Figure 2.6: Staged Configuration by eliminating features

Czarnecki et al. define the definition of specialization in [CHE04] as: " The specialization process is a transformation process that takes a feature diagram and yields another feature diagram, such that the set of the configurations denoted by the latter diagram is a subset of the configuration denoted by the former diagram." After a review of the definitions of configuration and specialization, we may have a question, what is the difference between the configuration and the specialization? A brief explanation of differences between specialization and configuration is that, a full specialization of feature diagrams denotes only for one configuration. Staged configuration is a form of configuration from the most specialized feature diagram in the specialization sequence [CHE04].

When we want to perform the configuration of SPL, we have the following methods: directly deriving a configuration from a feature diagram, staged configuration from specialized feature diagram in the specialization sequence, deriving of the configuration after the specializing a feature diagram to a fully specialized feature diagram [CHE05a, CHE04, CHE05b]. The methods of directly deriving a configuration from a feature diagram and configuration after the specializing a feature diagram to a fully specialized feature diagram are two extreme methods in staged configuration.

The steps of specialization are folliwing [CHE04]:

- Refining a feature cardinality
- refining a group cardinality
- Removing a grouped feature from a group
- Selecting a grouped feature from a group
- Assigning a value to an attribute which only has been given a type
- Cloning a solitary subfeature
- Unfolding a feature diagram reference

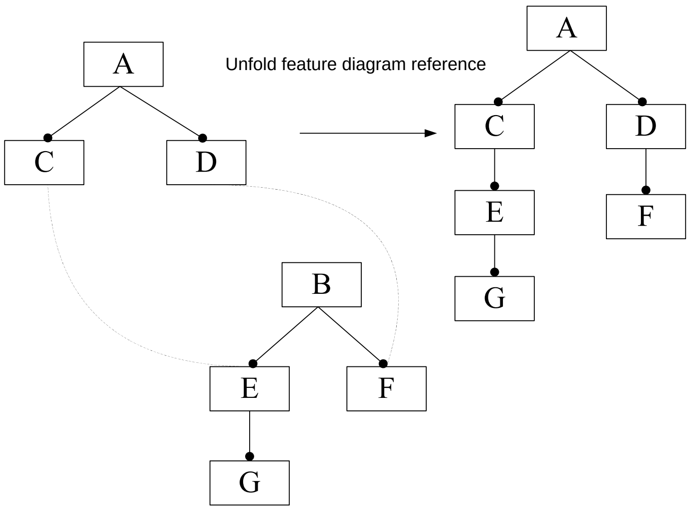


Figure 2.7: Unfolding a feature diagram reference [CHE04]

Figure 2.7 shows the unfolding of a feature diagram reference. Feature C references feature E and feature D references feature F. So if we want to select feature C, we also has to select feature E and G. If we want to select the feature D, feature F also has to be selected. That means C and D depend on feature E, F and G. So we can draw the feature diagram like the right feature diagram shown in Figure 2.7. That is the dependent SPLs, SPL A uses the functionality supported by SPL B. SPL is well suited for the similar programs, but if we have many SPLs, the dependencies between SPLs will occur, so we have to develop an approach to model dependencies between SPLs. In this thesis our main task is using a case study to modeling and implementation of the dependent SPLs.

2.1.3 Feature-Oriented Programing

Feature-Oriented Programming (FOP) encapsulates features in feature fragments and groups of fragments. We can easy reuse feature fragments and develop feature fragments in a process of stepwise refinement. Fragments are composed at compile time in order to form the classes and hierarchies of a final application. An FOP approach is useful for the development of SPL[1].

In this section we will have a review of two models of the FOP: GenVoca and Algebraic Hierarchical Equations for Application Design (AHEAD). And then we will have a review of a technique of FOP which called MiXin layers. For the implementation we have the Feature-Oriented C++ (FC++). And we will also have a brief review of Feature IDE and feature-Oriented Programming in Java.

[1]http://www.cse.fau.edu/ mike/fop.html

GenVoca GenVoca is a FOP model which describes how the code representation is expressed by an equation, e.g., an individual program can be expressed by incrementally adding details [BSR04]. Like our sensor network, we can incrementally adding encryption methods, e.g., DES and AES, in order to satisfy the requirement of users. As mentioned in the section of motivation that dependencies and constraints exist among dependent SPLs. Although the constants and functions of GenVoca is un-typed, the constrains of typing exist as design rules [BSR04]. Design rules are helpful for the compositions, which capture syntactic and semantic constraints among dependent SPLs.

- Constant of GenVoca is a set of classes which represents a base individual program, e.g., the constant b, which means the program with feature b.
- Function of GenVoca is a set of classes and refinements of those classes. A refinement is a function which takes a program as input, then produce a feature-oriented program as output, e.g., the function k(x), which means that we add feature k to the program.

We can introduce new details, e.g., data members, methods and constructors, to a base class, and we also extend and override the methods and constructors which are existing of that base classes [BSR04]. Different equations define a family of applications. If an application has many features, then we will use the equation to express the multiple features, we can select the features or functionalities that we need from the equation. Such as: app1 = i(j(f)), that means the application app1 has features i, j and f. Then we can determine the features from the equation, and a feature model is a set of constants and functions, and we can use the constants and functions to build our product lines [BSR04].

Mixins Mixins provide an easy way to add new data members, methods and constructors of an existing class. Minxins extend the methods and constructors which existing of its super class. The super class is specified by a parameter normally. Mixin do not inherit the constructors of its super class and a mixin does not assume the name of its super class, so Mixins are just like the class refinement [BSR04].

Figure 2.8 shows the linear refinement chains, which are common in this method of implementation. As shown in the Figure 2.8, the first layer is constant or base program which encapsulates three classes. The second layer is the function j which refines two classes and adds a new class. That means, function j refines classes a_i and c_i. The application of function k to j(i) results in the refinement of one class c_j. The composition k(j(i)) produces four classes a, b, c, d. Each class is refined by the feature i, j, k. We can express the refinement like: class a extends $a_j(a_i)$. We represent the class refinement of mixins as functions, such as in the Figure 2.8, minxin $a_j()$ is a function,

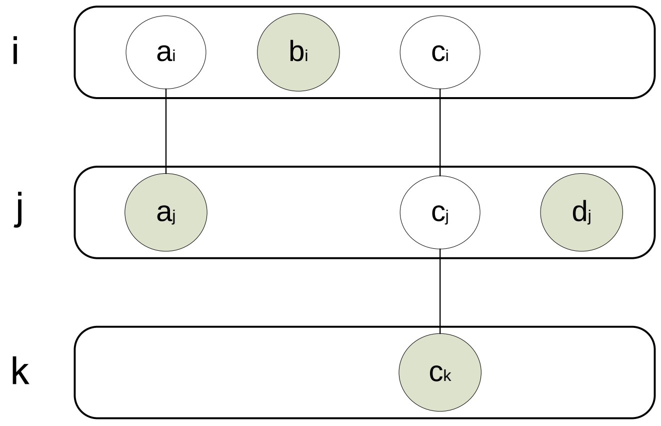


Figure 2.8: Implementing refinements by mixin inheritance [BSR04]

we apply this function to the base class a_i. We use the expression of $a_j(a_i)$ to describe the linear refinement chain of the classes. In this example only the classes a_j, b_i, c_k, e_j are instantiated, the other class are never instantiated. In the linear refinement we can find out that only the classes of terminal are instantiated, the classes of non terminal are not instantiated of the refinement chains [BSR04].

AHEAD We have known that GenVoca expressed the code representation of an individual or base program as an equation. The other model of FOP, which we will present here, is Algebraic Hierarchical Equations for Application Design, shortly we called it as AHEAD. We will have a brief review of the constants and functions and discuss how the constants, functions and compositions of AHEAD are represented. An arbitrary number of programs are expressed by AHEAD, and the representations are nested sets of equations [BSR04]. The definitions of constants and functions are the same as GenVoca, the base artifacts or programs are constants, and the refinements of those artifacts are functions. We model the refinement of the artifacts or constants as a series of functions or refinement. We can use the Minxin layer to model this refinement chains, and we can clearly have a look at the constants and functions. The Figure 2.12 shows our graphical notation for a GenVoca constant which encapsulates from base artifacts. We can use a set of constants to express the constant mathematically. For the composition, we write h(f) to express the composition of the artifacts h and f. Instead of this way to denote the composition, we can write the h(f) as $h \bullet f$.

Figure 2.9 shows the graphical notation and we will show here its AHEAD expression here: $h \bullet f = \{a_h \bullet a_f, b_f, c_h \bullet c_f, d_h\}$

In comparable with the graphical denotation like Figure 2.9, the AHEAD expression is: $a_h \bullet a_f$ is the refinement chain for artifact a, b_f and d_h are not refined in the refinement chains, so they are not changed from their original definitions, and the $c_h \bullet c_f$ is the

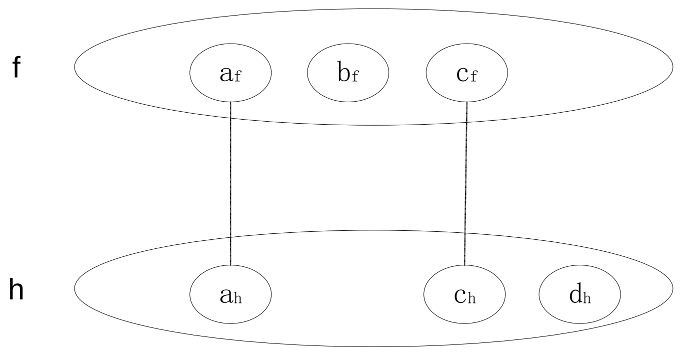


Figure 2.9: Expression and refinement chains

refinement chain for the artifact b [BSR04].

In this section, we discussed the two models of the FOP (i.e., GenVoca and AHEAD), and one popular technique of FOP called Mixins, one minxin layer express one feature which shows the refined classes and the added classes. The FC++ is a programming language which is the extension of the C++. We can also use the FOP in java, but the code files which composed by AHEAD tools, e.g., jampack or mixin, are not the pure java, but the Jak files. Then we can use the jak2java as the second step to make a Jak file into the java file. For the FeatureIDE, it is just a plug-in for an Eclipse [LAM05]. We can use this plug-in to model the feature diagrams in Eclipse, and the expression of the diagrams will be generated dynamically, and we can choose the features that we need from the equation, and produce the real system for the customer who has special requirements.

2.2 Dependent Software Product Lines

SPLs are used to develop software which can be customized to different use cases [CE00]. Sometimes multiple SPLs are integrated into one larger SPL to fulfill a special task [Omm02]. One SPL may use functionalities or components provided by other SPLs, the involved SPLs are dependent on each other. If we want to configure one SPL which depends on other SPLs, the configuration of other SPLs is also needed. When creating products from these SPLs all created SPL instances have to be compatible. In SPLE an SPL instance is a concrete product which derived from SPL. A concrete product can be derived by selecting the needed features from an SPL based on the implementation of SPL. The created SPL instance might be a component, a program or a collection of programs. So the configuration of each concrete SPL instance has to be defined. Staged configuration corresponds to specialization of feature models. We can remove unneeded functionalities using staged configuration. A user who configures an SPL which depends on other SPLs is usually interested in configuration decisions of the domain, e.g., in sensor network, users are only interested in configuration of communication protocols they use, they are not interested in the configuration of the product line which provide the functionality of communication protocols. So the underlying SPLs should be automatically configured to match requirements of enclosing SPL. Only functionality which users are interested has to be configured manually. We can define constraints in order to show the dependencies between dependent SPLs. Constraints can be defined not only on the domain level, but also on the instance level, if multiple products of one SPL are used, this is instance constraints [RSKuR08]. The concrete composition mechanism is dependent on the implementation technique.

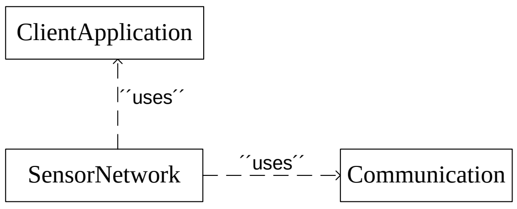


Figure 2.10: A Sensor Network SPL using a Communication SPL and Client Application SPL

An example is shown in Figure 2.10. A sensor network is developed as an SPL SensorNetwork and uses communication protocols provided by Communication SPL. Using Bluetooth communication protocol in sensor networks requires the Communication SPL to provide this functionality of Bluetooth. If the user needs the functionality of Wi-Fi in sensor networks, then the Communication SPL has to provide this functionality of Wi-Fi. Users are only interested in the configuration of SensorNetwork SPL, and the underlying SPLs should be automatically configured, like Communication SPL and Client Application SPL in sensor network systems. In Figure 2.10 constraints are defined on the domain level. Using domain constraints, all dependencies within an SPL and between dependent SPLs can be modeled.

We will use an example to explain dependent SPLs. A, B and C are three SPLs. A is the underlying SPL of B and C. That means, B and C need functionalities supported by A to fulfill special tasks. B needs the functionality of encryption method AES, and C needs the functionality of encryption method DES, we need to configure A by selecting features AES and DES. As shown in Figure 2.11.

SPL instance is a concrete product which derived from SPL. In Figure 2.12, PDA and Laptop are instances of ClientApplication SPL, Bluetooth and Wi-Fi are two instances of Communication SPL. If multiple products of one SPL are used, then the constraints between SPL and instances are needed. In this example, if a sensor network software system implements the communication protocol among sensor nodes using Bluetooth,

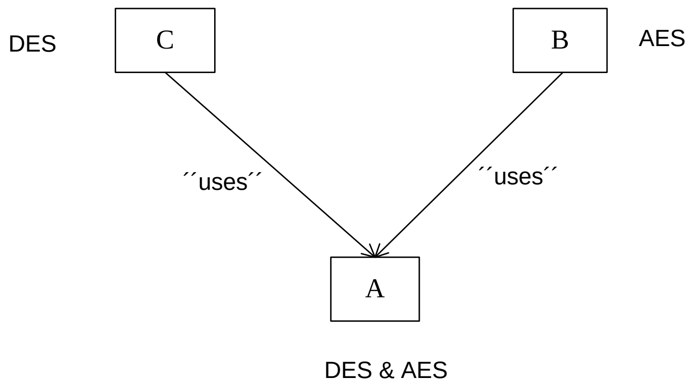


Figure 2.11: Dependent SPLs

and another sensor network software system implements the communication protocol among sensor nodes using Wi-Fi, then constraints between SensorNetwork SPL and instances of Communication SPL have to be considered. PDA and Laptop are instances of ClientApplication SPL, constraints between SPL SensorNetwork and instances of ClientApplication SPL have to be defined. Constraints between SensorNetwork SPL and clientApplication SPL cannot describe those dependencies. In Figure 2.12 constraints are defined on the instance level.

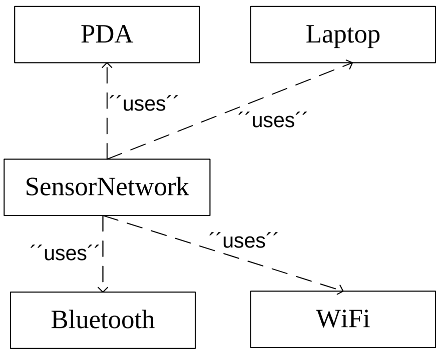


Figure 2.12: A Sensor Network SPL using different instances of Communication SPL and Client Application SPL

We explained the definition of dependent SPLs. Feature models cannot describe the constraints and dependencies between dependent SPLs and concrete SPL instances, Marko Rosenmueller et al. presented an extension to current SPL modeling based on class diagrams that allows us to describe SPL instances and dependencies between dependent SPLs and concrete SPL instances [RSKuR08]. We will use this approach to model our sensor network scenarios in the next chapter.

2.3 Sensor Networks

A sensor network consists of a large number of sensor nodes which are capable of sensing the environment. Nearby sensor nodes can communicate with each other using communication protocols. Sensors are generally equipped with data processing and communication capabilities, e.g., BTnode is a sensor node which supports autonomous wireless communication and computing platform based on a Bluetooth radio. BTnode consists of Microcontroller, Memories and Bluetooth subsystem[2]. A sensor network usually includes sensor nodes, sink nodes and the management node. A large number of sensor nodes randomly deployed in the field of the monitoring, through self-organization can constitute a network. Sensor node can monitor the data from the field and down to the other sensor node by the hop-by-hop transmission. During transmission monitoring data may be handled by multiple nodes, and will be sent to a sink node after multi-hop routing, finally arrive via the Internet or satellite to the management node. Users through the management node of the sensor network configuration and management, release monitoring tasks and the collection of monitoring data. Sensor node is usually a tiny embedded system, and his ability to deal with, storage capacity and communications capability is relatively weak, and the adoption of portable battery-powered energy limited. Functional point of view from sensor networks, each sensor node has the traditional terminals and routers dual function, apart from local information collection and data processing, but also to other nodes to forward the data storage, management and integration. The processing capability, storage capacity and communication capacity of sink node is relative strength of sensor nodes, and it has ability to connect with sensor networks and other external networks, such as Internet. Sink node can implement communication protocol conversion between the two types of protocol stacks, and release management node monitoring tasks, and receive forward data to the external network. Sink node can be both an enhancement of the sensor nodes, there is sufficient energy supply and more memory and computing resources may also be a function not only to monitor the wireless communication interface with the particular gateway device [Mah07].

In sensor networks, communications are usually trigged by queries or events [SMZ07]. Sensor nodes can communicate with each other using broadcast or point-to-point. All nodes in sensor network do not have a global ID such as IP number. The security of sensor networks is more limited than conventional wireless networks, So we should use encryption methods to guarantee the security of sensor networks. On the Internet, the network equipment uses network IP address as the unique ID. Resource Locator and information transmission depends on the terminals, routers and servers, network equipment, such as the IP address. Sensor networks are task-based network, talk about from the sensor network sensor node does not have any meaning. Sensor network node

[2]http://www.btnode.ethz.ch/

uses the node ID as a logo. Node ID is required or not in the whole sensor network is dependent on the network communication protocol unique design. Users query the event of sensor networks, concern directly to the event notification to the network, rather than a notice to determine the number of nodes. Network access to the information designated after the case passed to the user. So sensor network is a data-centric network.

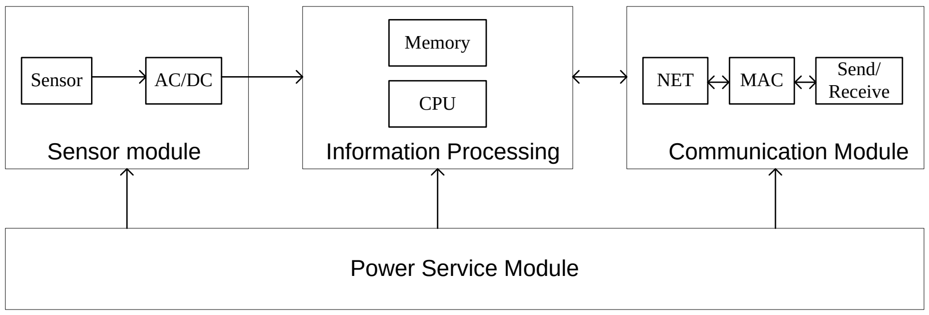


Figure 2.13: Sensor Network Node Architecture [SMZ07]

Sensor node consists of sensor module, information processing, communication module and power service module as shown in Figure 2.13. Sensor module is responsible for monitoring area information collection and conversion. Information processing is responsible for the management of the whole sensor node, storage and handling their data gathering or any other node of data. Communication module is responsible for communication with other sensor nodes. Power service module is responsible for the operation of the whole sensor network energy supplies [SMZ07].

Now we have many types of sensor network routing protocol, such as Flooding, Gossiping and SPIN (Sensor Protocol for Information via Negotiation) [SMZ07].

Flooding is a tradition of wireless communication routing protocol. In this routing protocol, each node has to accept the information from other nodes, and to send the information to neighbor node using broadcast. And so it went on, finally will send data destination node. But this protocol easy to cause the implosion and overlap of information, and wasting of resources. So in this routing protocol is put forward, based on Gossiping routing protocol [SMZ07].

Gossiping protocol spread information is by the way of random choice a neighbor node, the information in the same way the neighbor node random selection next neighbor node for the transmitting of information. The way to avoid the spread of forms of broadcast energy consumption. But its price is to extend the time of information transfer. Although this protocol on the solution to a certain extent the implosion of information, but still exists the phenomenon of overlapping information [SMZ07].

SPIN is a center with data of adaptive routing protocol. The purpose of this protocol is: through consultation between nodes, to solve the implosion phenomenon of Flooding

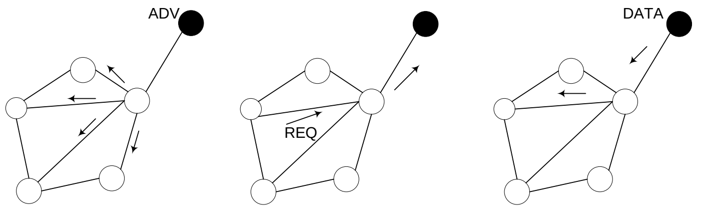


Figure 2.14: Three types of messages of SPIN routing protocol

and Gossiping protocols, but still exists the phenomenon of overlapping information. There are three types of messages of SPIN protocol, namely ADV, REQ and DATA as shown in Figure 2.14. ADV is used for data broadcast, when a node has data can be Share, we can use it for information broadcast. REQ is used for requesting to send data, when a node hope to accept data packets, then it will send REQ. DATA is the collected packets of sensors [SMZ07].

Bluetooth Bluetooth is a short-range wireless communications technology norms developed by the Bluetooth Special Interest Group (SIG) in 1998. It is a kind of wireless data and voice communications open standards, and the purpose is the way of wireless replacing cable interface. It has strong portability and can be applied to various communication occasions. Low power consumption and the harm to human body are the advantages of Bluetooth. Bluetooth uses spectrum frequency hopping technique. As a representative of the frequency hopping spread spectrum radios, it is alternative to broadcast radios in sensor networks. The strong anti-jamming capability of Bluetooth can increase the safety of information transmission. Bluetooth systems support peer-to-peer and point-to-multipoint communication. Each network of Bluetooth device called Piconet. In Piconet Bluetooth devices use the Way of master-slave to realize communication. Because of the physical addressing a Bluetooth device, at the same time, Piconet only activate eight equipments, i.e., one master and seven slaves in a Piconet. But different time, more the Piconets could constitute a scattering of overlapping network structure. Bluetooth communication and the radius of the effective output power: when the output power is 2 class (2.5 mW / 4dB), communication range is 15m; if the power to increase 1 class, 4mW / 20dB can make communication range to 100m. We can use Bluetooth to realize the communication of sensor nodes in sensor networks[3].

Bluetooth sensor network model is based on the principle of the neighboring network. Two near each other to a certain degree of Bluetooth sensors can spontaneously by the Bluetooth module to establish communication links. Bluetooth group network can have up to 256 Bluetooth devices to connect modules together to form Piconet. Among them,

[3]http://article.ednchina.com/Communinet/20071113082906.htm

a master node and 7 slave nodes in working condition, while the other nodes are in idle mode. Master node is responsible for control of asynchronous connectionless link (ACL) bandwidth. The slave nodes can only send data when being selected. Bluetooth specification allows connecting more Piconets together to form a Scatternet. In this case some devices acting as a bridge by playing the master role and the slave role in one Piconet at the same time. Bluetooth communication protocol has C/S architecture. The client is the device which initiates the connection. The server is the device who receives the connection. In Bluetooth sensor networks, if an event from a sensor occurs, the sensor has only to send this event to its master. And its master has to send this event to its master till the event reached the base station [Zen07].

In this section, we made a brief description of sensor networks. Wireless sensor network usually consists of thousands of sensor nodes. Therefore, the wireless sensor network routing is particularly important. Flooding is a tradition of wireless communication routing protocol. Gossiping protocol spread information is by the way of random choice a neighbor node. SPIN is a center with data of adaptive routing protocol. Finally, we made a review of Bluetooth. Bluetooth is the communication protocol in our sensor network scenarios.

2.4 Summary

In this chapter, we introduced Software Product Line Engineering. Domain Engineering and Application Engineering are two parallel parts of SPLE. Next, we introduced FODA and feature models. We explained FODA in two parts: definition of domain analysis which is the first step of Domain Engineering, and explanation of the feature-oriented part of FODA. Then, we introduced staged configuration using feature models. The process of specifying a family member can be performed in stages, where each stage eliminates some configuration choices. Then, we made a brief review of FOP and the two models of FOP, i.e., GenVoca and AHEAD, and made a review of a technique of FOP called MiXin layers. For the implementation we have FeatureC++ and Jak. FeatureIDE is a plug-in of Eclipse. Furthermore, we introduced dependent SPLs. Sometimes multiple smaller SPLs are integrated into one larger SPL to fulfill special tasks [Omm02]. One SPL may reuse functionalities or components provided by other SPLs, the involved SPLs are dependent on each other. If we want to configure one SPL which depends on other SPLs, the configuration of other SPLs is also needed. We can use feature models to describe the dependencies and constraints between dependent SPLs on a domain model level. An SPL instance is a concrete product which derived from SPL. If there are dependencies between concrete SPL instances and SPLs, we cannot describe this dependencies and constraints using feature models, so we have to extend the feature models to describe the dependencies and constraints between concrete instances of SPL. In the next chapter, we will use sensor network scenario to describe modeling and implementation approaches of dependent SPLs. Finally, we made a brief description of sensor networks and the communication protocols in sensor networks.

Chapter 3

Modeling Dependent Software Product Lines

In this chapter we use a sensor network scenario to describe dependent SPLs. FODA is used to analyze possible features in sensor network systems. The SPLs for a sensor network will be modeled using feature diagrams. Afterwards, two approaches for modeling dependencies between dependent SPLs will be described, i.e., feature model references (FM-References) and modeling instances of SPLs. FM-References can describe dependencies between dependent SPLs without taking instances into account. The approach of modeling instances for SPLs is an extension to current SPL modeling based on class diagrams [RSKuR08], which allows us to describe SPL instances, and describe dependencies between them. Finally, we will compare the two approaches for modeling dependencies between dependent SPLs.

3.1 Case Study: Sensor Network Product Line

In this section we use the concept of Domain Engineering and Domain Analysis to describe a sensor network product line. A sensor network consists of different sensor nodes and clients. There are two kinds of devices in sensor network scenarios, e.g., BTnode and Laptop. First, we need a product line for sensor nodes (SensorNetworkNode SPL), implementation of sensor functions, wireless communications and data storage capabilities. All possible features for sensor nodes are analyzed and described in a feature diagram (SensorNetworkNode). Second, we need a product line for client applications (ClientApplication SPL), which implements the communication between different clients and sensor nodes, and send queries to sensor networks. All possible features for different clients which access sensor networks are analyzed and described in a feature diagram (ClientApplication). It is easy to see that, both in SensorNetworkNode SPL and ClientApplication SPL we have to implement the functionality of communication. So we need a product line for communication (Communication SPL), which provides the functionality of communication for SensorNetworkNode SPL and ClientAplication SPL. A Communication SPL can provide the reuse of code. The code can be reused in different SPLs (SensorNetworkNode SPL and ClientApplication SPL). We can achieve functionalities of network communication. We will use the existing FAME-DBMS SPL to support the functionality of data storage[1]. Then, we will explain all SPLs in sensor network software systems.

3.1.1 Product Lines for a Sensor Network

In this section we describe needed product lines for a sensor network.

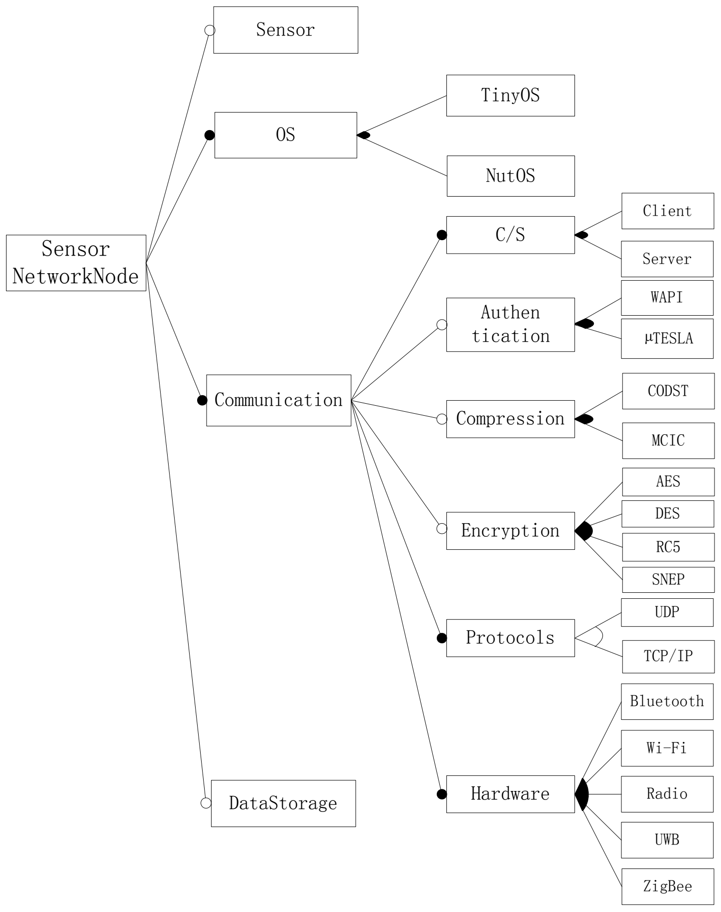


Figure 3.1: Feature Diagram for Sensor Network Nodes

SensorNetworkNode SPL In Domain Analysis the software of a sensor network node can be considered as an SPL, which might have four features, i.e., OS, Sensor, DataStorage and Communication as shown in Figure 3.1. A mandatory feature of every

[1]http://fame-dbms.org/

wireless sensor network node is that it communicates with other nodes through its communication capabilities. The optional features of every wireless sensor network node are sensor and data storage capabilities. So we have two optional features, i.e., Sensor and DataStorage, and two mandatory features, i.e., Communication and OS. If the optional feature Sensor is selected, then the nodes will sense data in the monitoring field. If the optional feature of DataStorage is selected, then the nodes will store the sensed data, in this case we will use the functionalities provided by FAME-DBMS SPL. In SensorNetworkNode SPL the feature Communication is mandatory, and it has sub-features as shown in Figure 3.1. The functionalities of communication are provided by the Communication SPL. The operating system of sensor network nodes can be TinyOS or NutOS. The storage capacity of Wireless sensor network node is limited and general operating system cannot be used for sensor nodes, e.g., Windows and Linux, so we have to use the embedded operating system on sensor nodes. For example NutOS can be used on BTnode[2].

The mandatory features of Communication in SensorNetworkNode SPL are C/S (Client/Server), Protocols and Hardware. Each node independently collects data, and sends it to the client. When communication occurs, a client connects with a sensor network, and sensor network node waits for connection. The communication protocols in SensorNetworkNode SPL is mandatory, we have two alternative features UDP and TCP/IP. The Hardware feature is also mandatory. We have Bluetooth, Wi-Fi, Radio, UWB and ZigBee. We will describe these features in Communication SPL.

In sensor networks, adversaries can listen to data, intercept data, inject data and alter transmitted data. Sensor networks have security risks, so we can use the Encryption technologies to protect sensor networks, e.g., packets are encrypted by the sender and decrypted by the receiver. In SPL SensorNetworkNode, we have an optional feature Encryption, which provides encryption methods, such as AES, DES, RC5 and SNEP that can be used in sensor networks. SNEP is sensor network protocol provides data authentication, encryption and refresh between the sender and receiver.

Data in sensor networks might be compressed, because of sensor node's storage capacity constraints and high power consumption for transmitting data. In SensorNetworkNode SPL, we have an optional feature Compression, which provides the compression functionalities in sensor networks. We have two alternative sub-features of Compression feature, i.e., MCIC and CODST. MCIC (Multi-node Cooperative Image Compression) uses low complexity and high compression performance of LBT image compression. CODST is a compression technique based on curve simulation and is proposed to compress streaming data collected by each sensor node. The compressed streaming data are recovered in the base station.

[2]http://www.btnode.ethz.ch/

Broadcast authentication is a critical security service in sensor networks. In SensorNetworkNode SPL, we have an optional feature Authentication, which provides the functionality of authentication. It allows a sender to broadcast messages to multiple nodes in an authenticated way. We have two alternative features in Authentication, i.e., μTESLA and WAPI.

ClientApplication SPL We need a product line for client applications, which implements the communication between users and sensor networks. So we develop a client application accessing the sensor network as an SPL ClientApplication, which supports different client hardware, such as Laptop and PDA, to communicate with sensor network nodes. The software is running on sensor nodes, and the client software access the network. So we have three features in ClienApplication SPL: OS, Types and Communication. The feature Types is an optional feature, which consists of two alternative sub-features, i.e., Laptop and PDA. The operating system of clients can be Windows or Linux, so we have a feature OS in ClientApplication SPL. We also have a mandatory feature Communication, which consists of different communication protocols and encryption methods etc. The Communication feature supports the communication between clients and sensor network nodes.

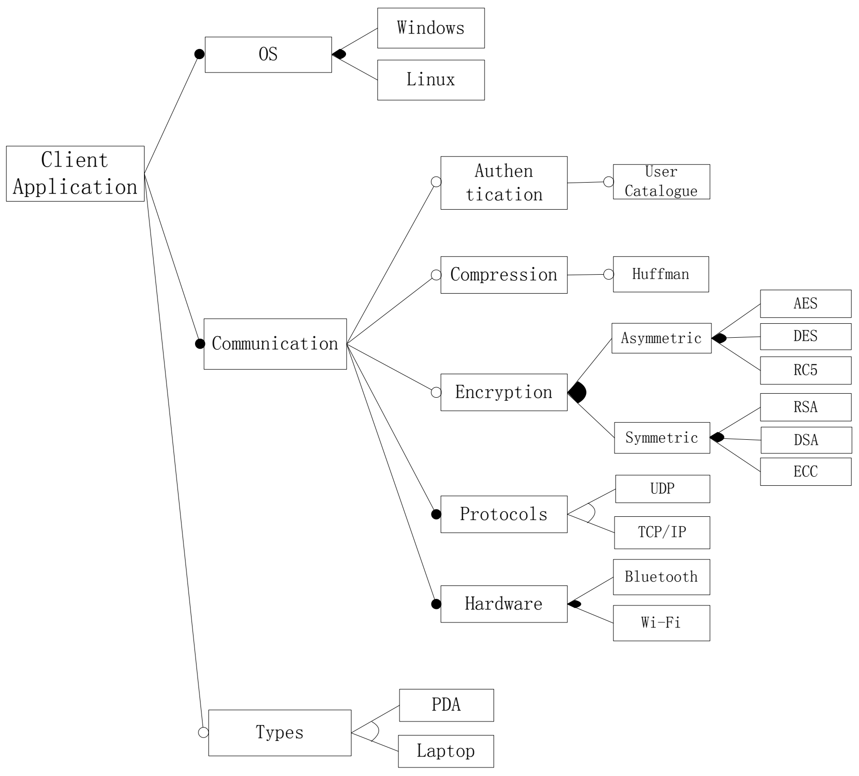


Figure 3.2: Feature Diagram for Client Applications

As shown in Figure 3.2 the mandatory features of Communication in ClientApplication SPL are: Protocols and Hardware. Communication protocol between a client and access node can be TCP/IP (Transmission Control Protocol/Internet Protocol) or UDP (User Datagram Protocol). Feature Hardware consists of two sub-features: Bluetooth and Wi-Fi.

The optional features of Communication in ClientApplication are: Authentication, Compression and Encryption. The feature of Authentication has an optional feature UserCatalogue, we can use the user ID and password to implement the functionality of authentication in client application system. The feature Compression has an optional feature Huffman, which is a compression method. Data in a client application might be compressed, because of the storage capacity constraints of sensor node and high power consumption for transmitting data. The feature Encryption has two sub-features: Symmetric and Asymmetric. Encryption can protect the security of transmitting data between client and sensor node.

Data encryption technology is divided into two categories, namely, symmetric encryption(Private key encryption) and asymmetric encryption(Public key encryption). Advanced Encryption Standard (AES), Data Encryption Standard (DES) and RC5 are typical encryption methods of symmetric encryption. Digital Signature Algorithm (DSA), Rivest Shamir Adleman (RSA) and Elliptic Curves Cryptography (ECC) are typical encryption methods of asymmetric encryption.

Dependencies between nodes of sensor network and client applications may exist to ensure a valid sensor network as a whole. Communication among sensor nodes requires the same communication hardware, for example Bluetooth. But the communication between the access node and client application may need Bluetooth or Wi-Fi to communicate with each other.

In ClientApplication product line, we can select the optional features shown in Figure 3.2 to configure different client applications which have different communication protocols. For example, one client may use AES as encryption method to interface with sensor networks, another client may use DES. So in this case, the ClientApplication and SensorNetworkNode SPLs have to provide AES and DES encryption method to fulfill the communication between both client applications and sensor nodes. So constraints may exist between ClientApplication SPL and the underlying SPL, e.g, a communication product line.

Communication SPL As mentioned above, we have multiple SPLs in our sensor network scenario. Because we have two kinds of devices in the sensor network scenario, we need the SensorNetworkNode SPL to develop software for the sensor node, and the ClientApplication SPL to develop software for the client which accesses sensor networks. The Communication feature is used in SensorNetworkNode SPL as well as in the ClientApplication SPL. Thus, we need a Communication product line to provide the communication functionality, taking into account the issue of code reuse.

The Communication SPL has a mandatory feature Operating System, which not only provides TinyOS and NutOS used by sensor nodes, but also provides Windows and Linux used by client applications. The mandatory feature Hardware provides all hardware used in sensor nodes and client applications. All features in Communication SPL are extracted features from SensorNetworkNode SPL and ClientApplication SPL. As shown in Figure 3.3, we have three optional features, i.e., Encryption, Compression and Authentication. The feature Encryption implements the functionality of encryption, not only for sensor nodes, but also for clients. So communication functionalities used in SensorNetworkNode SPL and ClientApplication SPL have to be provided by the Communication product line.

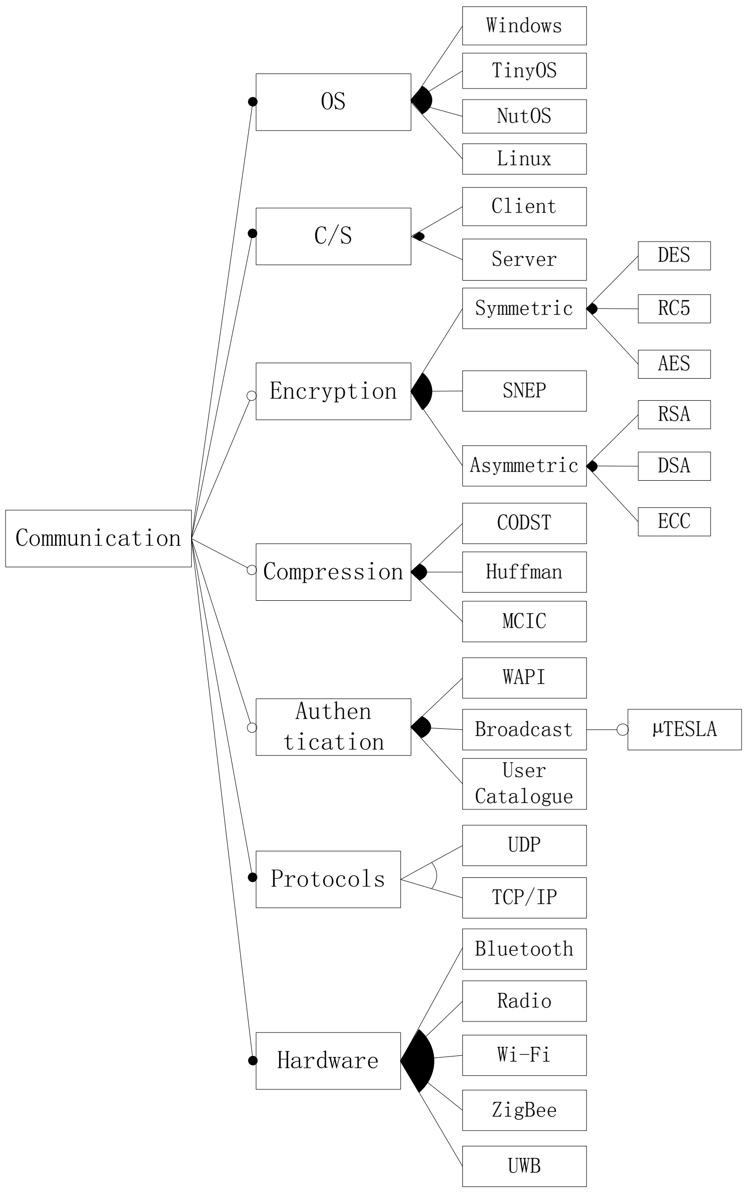


Figure 3.3: Feature Diagram for Communication

FAME-DBMS SPL As mentioned above, in SensorNetworkNode SPL, we use a FAME-DBMS SPL for data storage. SensorNetworkNode SPL is dependent on FAME-DBMS SPL. If we need a sensor node to store the sensing data, the functionality of data storage has to be provided by FAME-DBMS SPL.

In FAME-DBMS SPL, we have four mandatory features: OS-Abstraction, Buffer-Manager, Access and Storage. The OS-Abstraction feature hides platform dependent implementation. The OS-Abstraction feature has three alternative sub-features: NutOS, Win32 and Linux. The FAME-DBMS cannot only be used on embedded systems, such as NutOS, but also on the platform of windows and Linux. The BufferManager feature is used for page buffering and management of used and free pages. The memory allocation for page buffering in FAME-DBMS can be static or dynamic. The page-replacement algorithms of LRU (Least Recently Used) and LFU (Least Frequently Used) are used in FAME-DBMS for the management of used and free pages. The access feature provides API based access, e.g., put, get, remove and update data. The Storage feature of FAME-DBMS provides the functionalities to store and retrieve data. The B+Tree indexing are implemented for quickly search and add data, as shown in Figure 3.4. Gray features have further sub-features that are not displayed [RSS+08].

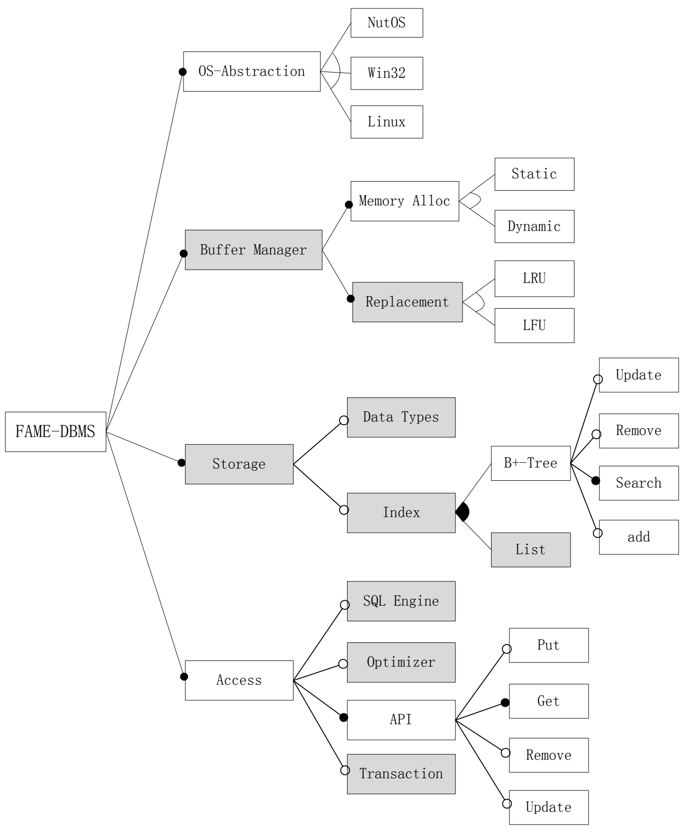


Figure 3.4: Feature diagram for FAME-DBMS [RSS+08]

3.1.2 Integrating Multiple Product Lines

As mentioned in background, SPLs can be reused as part of other SPL, and sometimes multiple product lines have to be integrated into one SPL to fulfill special functionalities. In our sensor network scenario, sensor networks are complex and distributed systems. So we can develop sensor networks as a product line which built from a number of heterogeneous SPL instances, i.e., SensorNetworkNode SPL instance and ClientApplication SPL instance. For example, a SensorNetwork SPL may consist of different sensor nodes, access nodes, and data storage nodes, each of them may be the instance of the SensorNetworkNode SPL. In sensor networks, we may have different access hardware, for example Laptops and PDAs, so each of them may be the instance of the ClientApplication SPL. The SensorNetwork SPL is not only an SPL from which we can create a program, but also a number of interacting programs. We can develop the software running on sensor nodes from SensorNetworkNode SPL, and the software of client accessing sensor networks from ClientApplication SPL. There might be no source code needed for SensorNetwork SPL, only the underlying SPLs contain program code. SensorNetworkNode SPL and ClientApplication SPL are the underlying product lines of SensorNetwork SPL.

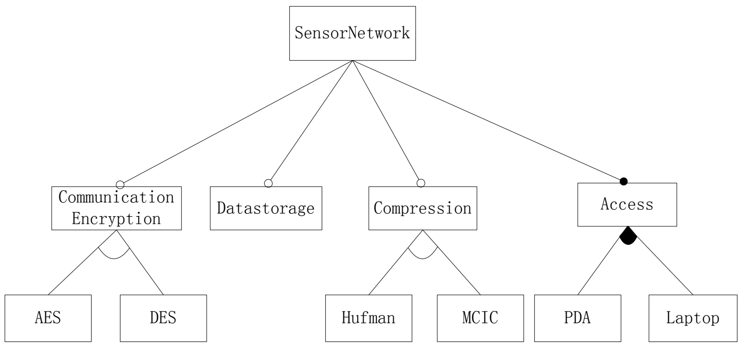


Figure 3.5: SPL for a Sensor Network

Figure 3.5 shows a sensor network product line, which consists of features Encryption, Compression, DataStorage and Access. The client applications in this sensor network can be PDA or Laptop. The DataStorage feature is an optional feature. Sensor network consists of many sensor nodes, access nodes and data storage nodes. The functionality of data storage is needed when we want to store data in sensor nodes. The sensor nodes are used for sensing, collecting and transmitting data. The access nodes are used for communicating with sensor nodes or client applications. These features in Figure 3.5 are visible to the end users. Users can select needed features to derive a sensor network system. All features in SensorNetwork SPL must be functionalities which end users are concerned. When creating a configuration, the user starts with an empty feature selection of the top level SPL, for example from SensorNetwork SPL. The underlying

SPLs of SensorNetwork SPL only added when they are needed. In the next section, we will describe the dependencies and constraints between dependent SPLs.

The SensorNetwork product line uses the functionalities provided by SensorNetworkNode product line and ClientApplication product line. SensorNetwork product line uses specialized SPLs ClientApplication and SensorNetworkNode to realize a real valid sensor network. In ClientApplication and SensorNetworkNode SPLs, we have many optional features. The selection of these optional features in such underlying SPLs depends on the other SPL, e.g., if a sensor network implements the functionality of data storage, then the DataStorage feature in the SensorNetworkNode SPL has to be selected. As shown in Figure 3.6, The SensorNetworkNode again uses the FAME-DBMS product line and Communication product line. DataNode uses FAME-DBMS product line to store data. If a sensor network has to implement the functionality of data storage, then the configuration of FAME-DBMS SPL is required. All nodes use the Communication product line to communicate with each other. The Communication product line is also the underlying product line of ClientApplication SPL, and provides special functionality for client applications. Figure 3.6 shows the dependencies between dependent SPLs in our case study sensor network scenario.

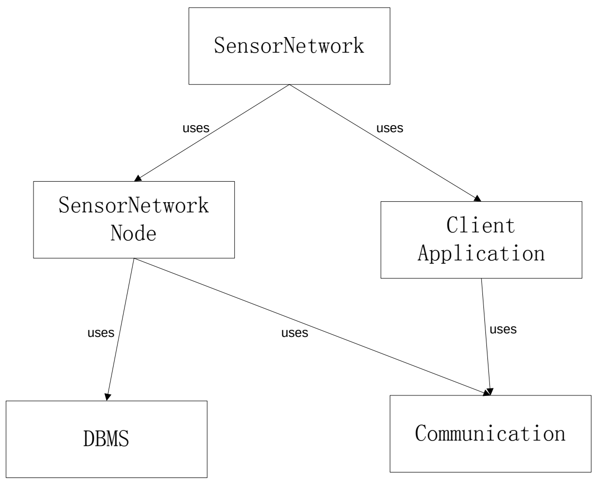


Figure 3.6: Dependent Software Product Lines

3.2 Using FM-References for Modeling Dependent Product Lines

In this section, we introduce an approach for modeling dependent SPLs. The involved SPLs may have dependencies and constraints. We will use FM-References to model the dependencies between dependent SPLs.

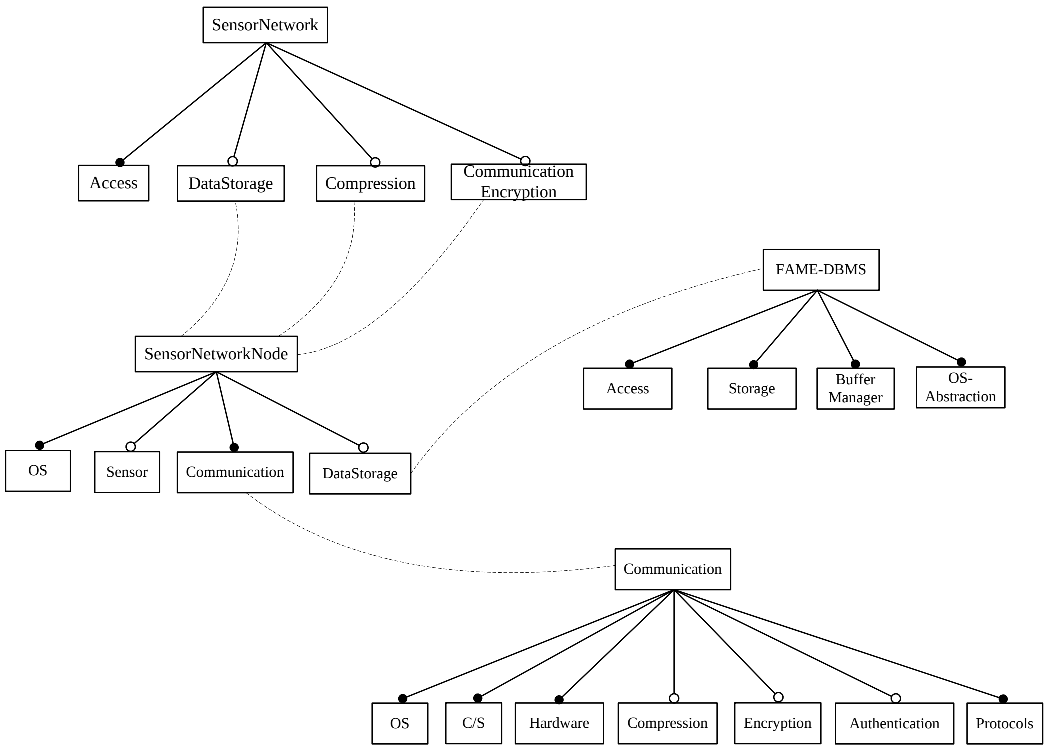


Figure 3.7: Using FM-References to model Dependencies between SensorNetwork Product Line and the underlying SensorNetworkNode, FAME-DBMS, and Communication Product Lines, adopt from [CHE04]

We have different types of composition rules, e.g., requires rules. When one feature requires another feature as condition, we use requires rules to show dependencies between features in multiple SPLs. SensorNetworkNode product line uses FAME-DBMS product line to fulfill the data storage functionality, and uses Communication product line to fulfill the communication functionality. If feature DataStorage is used in our SensorNetworkNode product line, FAME-DBMS product line is also required to provide the functionality of data storage. A sensor network consists of different sensor nodes, access nodes and data storage nodes, so if we don't want to store data in our sensor network, the whole FAME-DBMS SPL is not needed. We have an optional feature Encryption in our SensorNetworkNode product line. We can use encryption technology to form a secure wireless sensor networks. If feature Encryption is used in our SensorNetworkNode product line, also the feature Encryption of Communication product line is required. If feature AES is selected in our SensorNetworkNode product line, also the selection of feature AES in Communication product line is required. We have different communication hardware in sensor networks, such as Bluetooth, Wi-Fi, Radio. Depending on the used hardware, for example Bluetooth, the Communication product line has to provide the functionality of Bluetooth communication. Similarly, if feature Wi-Fi is used in our SensorNetworkNode product line, also the feature Wi-Fi of Communication product line is required. Figure 3.7 shows the dependencies between SensorNetworkNode and Communication product lines, and dependencies between SensorNetworkNode and FAME-DBMS product lines.

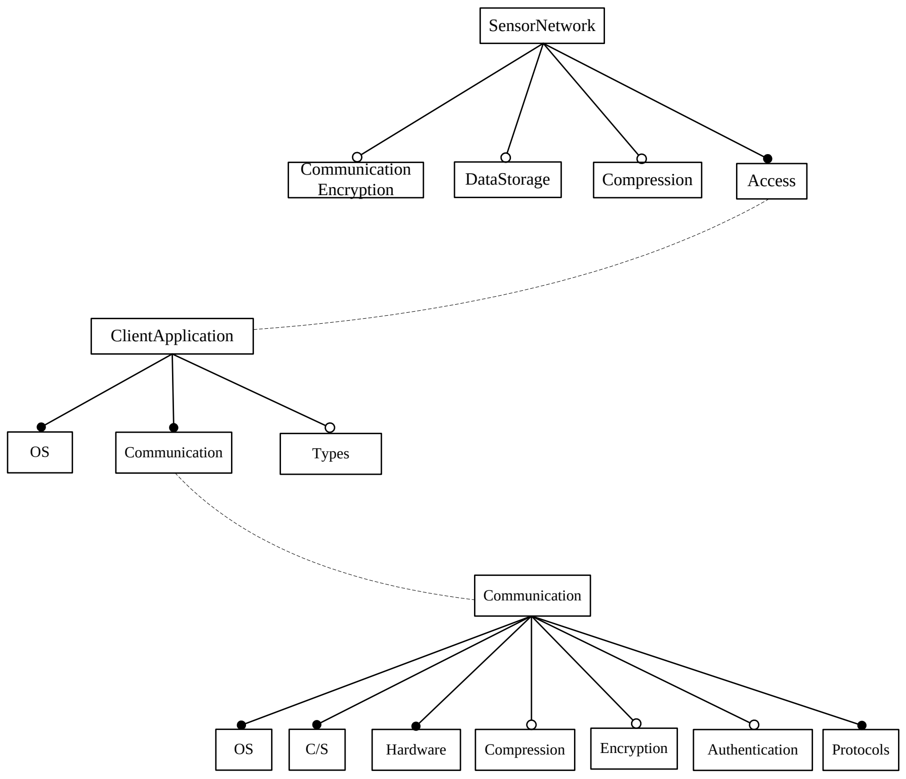


Figure 3.8: Using FM-References to model Dependencies between SensorNetwork Product Line and the underlying ClientApplication and Communication Product Lines, adopt from [CHE04]

Using FM-Reference we can also model the dependencies between ClientApplication product line and Communication product line, as shown in Figure 3.8. Communication product line is the underlying product line of ClientApplication product line. If we want to select one feature in ClientApplication product line, which requires the feature of Communication product line, we have to make sure that, the feature in Communication has to be selected. For example, we a client application use the TCP/IP communication protocols, the feature TCP/IP has to be selected in Communication product line.

Users of sensor networks only want to configure the SensorNetwork product line, so the references between dependent SPLs have to be defined. We cannot describe the dependencies and references between sensor nodes and data storage nodes, because in FM-Reference sensor nodes and data storage nodes are one SPL. Another example, a sensor node need the functionality of Radio to communicate with other sensor nodes, so the Communication product line has to provide the functionality of Radio to the sensor node, and an access node need the functionality of Bluetooth to communicate with a client application, e.g., PDA, so the Communication product line has to provide the functionality of Bluetooth to the access node. But in this case the sensor node and access node are one product line SensorNetworkNode, we cannot to describe the dependencies between the Communication product line and sensor node, we also cannot describe the dependencies between the Communication product line and access node. If we have another client application, for example Laptop, also want to access the sensor network, but this client application need the functionality of Wi-Fi to communicate with the sensor network, so in this case we cannot describe the references between Communication product line and PDA, and between Communication product line and Laptop, because PDA and Laptop is one product line ClientApplication. So we have to extend the feature diagram and to describe the dependencies on an instance level. In SPLE, creating an SPL instance means to derive a concrete product of a product line. We can use staged configuration to derive a concrete product line of an SPL step by step.

3.3 Modeling Software Product Line Instances

Using FM-Reference we cannot describe dependencies and constraints which only effect concrete instances of a product line. In SPLE, creating an SPL instance means to derive a concrete product of an SPL. We can derive different instances from SensorNetworkNode product line and ClientApplication SPL by selecting needed features. We can use Staged configuration to result in specialized SPLs, e.g., SensorNode SPL, AccessNode SPL, and DataStorageNode SPL. We can still create different sensor nodes from the specialized SensroNode SPL, similarly for specialized AccessNode SPL and DataStorageNode SPL. We can also derive different specialized SPL from ClientApplication product line using staged configuration, i.e., Laptop SPL and PDA SPL which needs different functionalities provided by Communication product line. As shown in figure 3.9, we have FAME-DBMS product line, SensorNetworkNode product line, Communication product line and ClientApplication product line in sensor network system. In this model SPL configuration is represented by inheritance and the relationship of uses is represented by aggregation [RSKuR08]. Czarnecki et al. compared SPLs to classes of Object Oriented Programming, and SPL configuration to class instances [CHE05b]. So based on the corresponding of Object Oriented Programming classes and SPLs, we can use class diagrams to model SPL compositions [RSKuR08]. Using a class diagram, our sensor network example can be modeled as shown in Figure 3.9. SPLs are represented by classes SensorNetwork, ClientApplication, Communication, SensorNetworkNode, and FAME-DBMS. Classes PDA and Laptop are specialized variants of the ClientApplication SPL that provide the functionalities of PDA and Laptop. Classes SensorNode, AccessNode and DataNode are specialized variants of SensorNetworkNode SPL that provide sensor nodes, access nodes and the functionality of data storage. Instances of SPLs which can be used by other SPLs are described using aggregation, e.g., members sensornode, accessnode, and datanode of class SensorNetwork represent instances of different specialized SensorNetworkNode SPLs. Members pda and laptop of class SensorNetwork represent instances of different specialized ClientApplication SPLs.

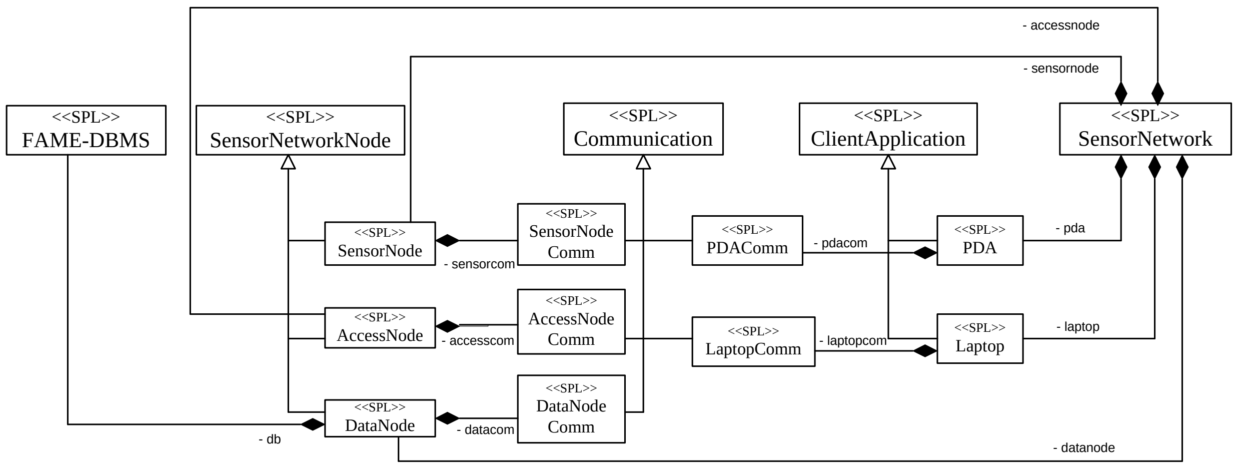


Figure 3.9: Dependencies between SPLs and instances [RSKuR08]

SensorNode SensorNode is a specialized SPL of SensorNetworkNode product line. For example, we select the optional feature Sensor of SensorNetworkNode product line, because in sensor network we need sensor nodes to sense and collect data from monitoring fields. The SensorNode instance also needs the functionality of communication to communicate with other sensor nodes to transfer data and monitoring tasks. In this case, we also derive an instance from Communication product line to provide only the communication functionality of sensor node, i.e., sensorcom. For example, sensor nodes need the hardware of Radio to communicate with other sensor nodes, so we have the select this feature from Communication product line.

AccessNode AccessNode is a specialized SPL of SensorNetworkNode product line. The optional features, e.g., DataStorage and Sensor, are not needed in this case. In sensor network Access node are used to communicate with client application, and collect the sensed data from sensor nodes. So we need the feature of Communication. The access node has to communicate with client application, and also have to communicate with sensor nodes. So the functionality of Communication is not the same as sensor nodes. For example, we need the functionality of Radio to communicate with sensor nodes, and we also need the functionality of Bluetooth to communicate with a PDA. We have to derive an instance from Communication product line to provide the communication functionality of access node, i.e., accesscom.

DataStorageNode DataStorageNode is a specialized SPL of SensorNetworkNode product line. We need the feature of DataStorage in DataStorageNode instance. We use the functionality of data storage to store data in sensor networks. FAME-DBMS product line provides the functionality of data storage, e.g., put and get data. Data storage nodes also need the functionality of communication to communicate with other nodes, for example sensor nodes and access nodes. So we have to derive an instance from Communication product line to provide the communication functionality of data storage node, i.e., datacom.

Laptop Laptop is a specialized SPL of ClientApplication product line. Laptop may need the hardware of Wi-Fi to access sensor networks, so the Communication product line has to provide this functionality. But we cannot describe constraints between ClientApplication product line and Communication product line, so we have to derive a concrete product and to describe the constraints. So we have to derive an instance from Communication product line to provide the communication functionality of Laptop, i.e., laptopcom.

PDA PDA is a specialized SPL of ClientApplication product line. We select the feature of PDA in this case. PDA also needs the functionality of communication to access sensor networks. For example, PDA need the functionality of Bluetooth to communicate with access nodes, so the underlying SPL Communication has to provide this functionality. A PDA can also need the functionality of Wi-Fi to communicate with PC. So we have to derive an instance from Communication product line to provide communication functionality of PDA, i.e., pdacom.

We can avoid constraints between SensorNetwork SPL and ClientApplication SPL which needed to define the different variants PDA and Laptop by using specialized variants. We only have to refer to the specialized variants, e.g., PDA and Laptop, and can reuse the configuration of the specialized SPLs in other SPL compositions. For example, SensorNetwork SPL can reuse the configuration of the specialized PDA SPL. We can also avoid constraints between SensorNetwork SPL and SensorNetworkNode SPL that needed to define the different variants SensorNode, AccessNode, and DataNode. SensorNetwork SPL can reuse the configuration of the specialized SensorNode SPL. We have defined different variants of Communication SPL, i.e., SensorNodeComm, AccessNodeComm, DataNodeComm, PDAComm, and LaptopComm. The configuration of these specialized

SPLs can be reused in SensorNetworkNode SPL and ClientApplication SPL, as shown in Figure 3.9.

Table 3.1: Domain Constraints

domain constraints
(1) SensorNetwork.MCIC ⇒ SensorNode.MCIC
(2) SensorNetwork.AES ⇒ AccessNode.AES
(3) SensorNetwork.DES ⇒ SensorNode.DES
(4) SensorNetowrk.Huffman ⇒ AccessNode.Huffman
(5) SensorNetwork.PDA ⇒ AccessNode.Bluetooth
(6) SensorNetwork.Laptop ⇒ AccessNode.Wi-Fi

Table 3.2: Instance Constraints

instance constraints
(1) SensorNetwork.DataStorage ⇒ SensorNetwork.pda.Queries
(2) SensorNetwork.CommunicationEncryption ⇒ SensorNetwork.pda.AES
(3) SensorNetwork.Compression ⇒ SensorNetwork.dataNode.CODST
(4) SensorNetwork.Compression ⇒ SensorNetwork.pda.Huffman
(5) SensorNetwork.CommunicationEncryption ⇒ SensorNetwork.datanode.SNEP

We have defined domain constraints in the domain model to describe constraints between dependent SPLs. Domain constraints can also be used to define constraints that apply for all instances of a product line. Table 3.1 describes the domain constraints in our sensor network scenario. We provide additional constraints for describing constraints between specialized SPLs. For example, constraint `SensorNetwork.PDA` ⇒ `AccessNode.Bluetooth` means that feature PDA implies feature Bluetooth only in specialized variant AccessNode of SensorNetworkNode product line. Constraints are also needed for concrete SPL instances. Specialized variants client applications and sensor nodes are used by SensorNetwork SPL. Specialized variants of Communication SPL are used by specialized variants of SensorNetworkNode SPL and ClientApplication SPL to provide communication functionalities. Table 3.2 describes the instance constraints in our sensor network scenario. For example, constraints `SensorNetwork.DataStorage` ⇒ `SensorNetwork.pda.Queries` means that, if feature DataStorage is selected in SensorNetwork SPL, we enable feature Queries in instance pda of SensorNetwork SPL.

Conditional dependencies are also needed in the composition of multiple dependent SPLs. Table 3.3 describes the conditional dependencies in our sensor network scenario. For example, dependence `SensorNetwork.PDA` ⇒ `SensorNetwork.pda` means that, if

Table 3.3: Conditional Dependencies

conditional dependencies
(1) SensorNetwork.Laptop ⇒ SensorNetwork.laptop
(2) SensorNetwork.PDA ⇒ SensorNetwork.pda
(3) SensorNetwork.DataStorage ⇒ SensorNetwork.dataNode
(4) DataNode.DataStorage ⇒ DataNode.datacom
(5) DataNode.DataStorage ⇒ DataNode.db
(6) SensorNode.MCIC ⇒ SensorNode.sensorcom
(7) SensorNode.DES ⇒ SensorNode.sensorcom
(8) AccessNode.AES ⇒ AccessNode.accesscom
(9) AccessNode.Huffman ⇒ AccessNode.accesscom
(10) AccessNode.Bluetooth ⇒ AccessNode.accesscom
(11) AccessNode.Wi-Fi ⇒ AccessNode.accesscom

we select feature PDA in SensorNetwork product line, then the feature PDA in ClientApplication product line also has to be selected. Dependence `SensorNetwork.Laptop` ⇒ `SensorNetwork.laptop` means that, if we select feature Laptop in SensorNetwork product line, then the feature Laptop in ClientApplication product line also has to be selected. Dependence `SensorNetwork.DataStorage` ⇒ `SensorNetwork.datanode` means that, if feature DataStorage is not selected, there will be no instance of DataNode product line. Other SPLs that DataNode depends on are not needed to add to the configuration. In our example, this applies to FAME-DBMS SPL and DataNodeComm SPL which don't need to be configured.

3.4 Discussion

In contrast to specialization, it is simpler to define instance constraints, but specialization provides better reuse. As mentioned above, specialized SPLs can also be used in other SPLs. The constraints which we presented are requires or implies constraints. We can use Object Constraint Language to specify constraints. In our model all constraints have to define the concrete SPL name and SPL instance name. We want to configure the underlying SPL automatically, but it cannot be achieved so simple, this can be done using a configuration generator in further work. The new approach for modeling SPL instances is only used when we cannot describe constraints and dependencies between different SPLs. By modeling dependent SPLs we will use the combination of both approaches to describe dependencies and constraints between different SPLs.

Chapter 4

Implementation

AHEAD Tool Suite includes the Jak language which implements AHEAD for Java, and FeatureC++ which implements AHEAD model for C++. In this chapter we will use the Jak language to implement our case study sensor network scenario on Eclipse with FeatureIDE Plug-in.

We made a review of dependent SPLs in the background chapter, sometimes multiple SPLs are integrated into one larger SPL and dependencies between the constituent SPLs occur. When one SPL uses another SPL to implement a special functionality, the configuration of the involved SPL is needed, so we will describe that, how these dependent SPLs can be automatically configured according to existing constraints between different product lines. We have described the approaches of modeling dependent SPLs using sensor network in the previous chapter, and in this chapter we will describe the implementation approaches for dependent SPLs. We will develop small SPLs to simulate a sensor network for evaluation.

4.1 Sensor Networks

In this section, we describe the implementation of SPLs, e.g., Communication product line, and the implementation of SPLs which use other SPLs, e.g., SensorNetworkNode product line and ClientApplication product line which use the Communication product line to realize the communication functionality. As mentioned in the previous chapter, all features in our sensor networks have been analyzed and described using feature models. We have SensorNetworkNode SPL, ClientApplication SPL, Communication SPL, FAME-DBMS SPL, and an integrated SensorNetwork SPL. Sensor network program is an complex and distributed software system. SensorNetworkNode SPL is used to develop the program for different sensor nodes in sensor networks. ClientApplication SPL is used to develop different client applications interfacing with network nodes. Communication SPL provides communication functionalities for sensor networks, e.g., encryption, compression. FAME-DBMS provide data storage functionalities, when we want to store data in sensor networks. We implement SensorNetworkNode SPL, ClientApplication SPL, and Communication SPL. This chapter does not relate to FAME-DBMS SPL, data storage functionalities are not implemented in our sensor network systems. We implement part of features from SensorNetworkNode SPL, ClientApplication SPL, and Communication SPL, and describe the approach of implementing multiple dependent SPLs.

4.1.1 Dependent Software Product Lines

As described in background chapter, dependent SPLs mean that, SPLs can be reused as part of other SPLs and sometimes functionalities or components of multiple product lines have to be integrated into one product line. In our case study, Communication product line is reused in SensorNetworkNode SPL and Clietapplication SPL. Figure 4.1 shows a simplified version of the feature diagrams described in the previous chapter which we used for implementation. Both in SensorNetworkNode SPL and ClientApplication SPL we have a Communication feature, which references with the Communication SPL. We implemented three features in Communication SPL, i.e., UDP, Compression, and Encryption. That means, Communication SPL can provide functionalities of UDP, compression, and encryption for SensorNetworkNode SPL and ClientApplication SPL. We can derive different variants of Communication SPL, e.g., UDP, which realizes the communication between sensor nodes and client applications using UDP communication protocol. The code of Communication product line can be reused in SensorNetworkNode SPL and ClientApplication SPL.

Except Communication feature we also have two alternative features in SensorNetworkNode SPL, i.e., NutOS and TinyOS. In our implementation, Communication feature implements the communication functionality over Ethenet between sensor nodes and client applications. We can develop a complex software program using step-wise refinement from a simple software program by adding features incrementally. So, we can add features Compression and Encryption incrementally to develop a complex system which has the compression and encryption functionalities. We also have two alternative features in ClientApplication SPL (i.e., PDA and Laptop). Those features such as NutOS, TinyOS, PDA, and Laptop don't have special functionalities in our implementation, only screen output. We just want to use these implementations as an example to describe how to implement an SPL and multiple dependent SPLs. First, we will describe the implementation of Communication SPL. Feature refinement will be described in this section. Then we will describe the implementation of SensorNetworkNode SPL and ClientApplication SPL which use Communication SPL to realize special functionalities.

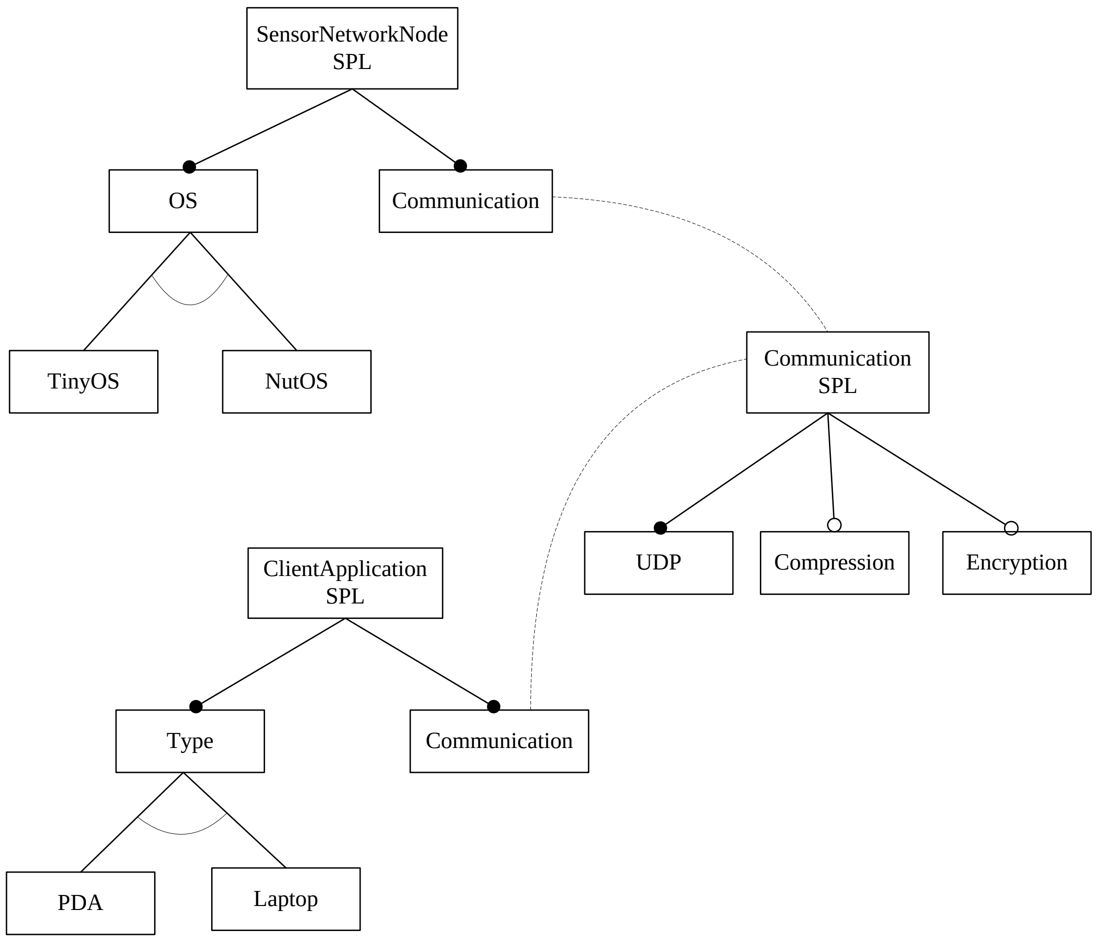


Figure 4.1: Dependencies between different Product Lines

4.1.2 Implementation of Communication Product Line

As introduced in the previous chapter, we analyzed all features of communication that can be used not only for SensorNetworkNode SPL, but also for ClientApplication SPL. We implement part of those features, e.g., UDP, Encryption, and Compression. Feature UDP provides the functionality of communication between sensor nodes and client applications using UDP communication protocol over Ethenet. Feature Compression provides the data compression functionality in sensor network systems. Feature Encryption is used for encrypted data storage and encryption of communication. The class diagram of Communication SPL is shown in Figure 4.2. We defined two different classes in Communication SPL, i.e., Comm and Listen. Class Listen is refined in other features by adding or removing functions. For example, we add a new function rot13 (a simple encryption method) in feature Encryption to implement the encryption functionality.

A feature refinement can encapsulate fragments of multiple classes. In Communication SPL, Communication feature encapsulates two classes, i.e., Class Comm and Class

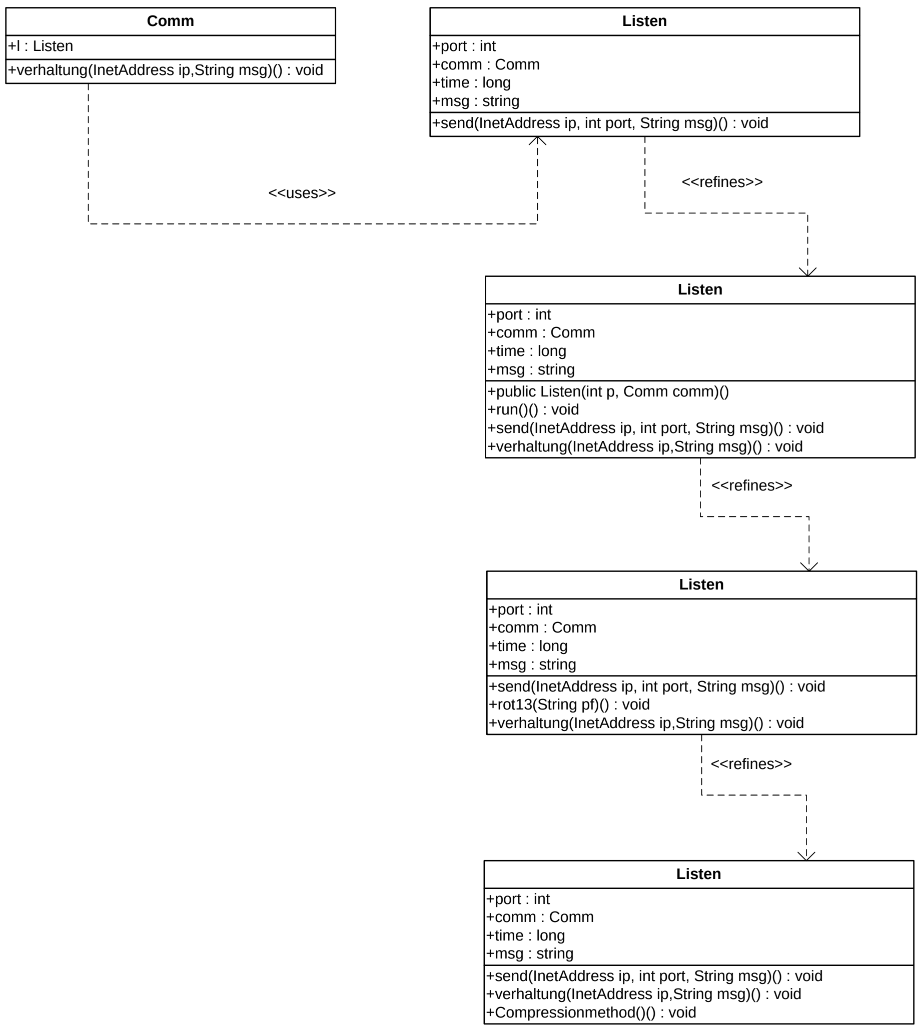

Comm
+l : Listen
+verhaltung(InetAddress ip,String msg)() : void
Listen
+port : int
+comm : Comm
+time : long
+msg : string
+send(InetAddress ip, int port, String msg)() : void
<<uses>>
<<refines>>
Listen
+port : int
+comm : Comm
+time : long
+msg : string
+public Listen(int p, Comm comm)()
+run()() : void
+send(InetAddress ip, int port, String msg)() : void
+verhaltung(InetAddress ip,String msg)() : void
<<refines>>
Listen
+port : int
+comm : Comm
+time : long
+msg : string
+send(InetAddress ip, int port, String msg)() : void
+rot13(String pf)() : void
+verhaltung(InetAddress ip,String msg)() : void
<<refines>>
Listen
+port : int
+comm : Comm
+time : long
+msg : string
+send(InetAddress ip, int port, String msg)() : void
+verhaltung(InetAddress ip,String msg)() : void
+Compressionmethod()() : void

Figure 4.2: Class Diagrams of Communication Product Line

Listen. Figure 4.3 depicts a package of two classes, Comm and Listen. Refinement Communication encapsulates fragments of Comm and Listen. Refinement UDP cross-cut Class Listen. The same holds for refinement Encryption and Compression. Composing refinements Communication, UDP, Compression, and Encryption yields a package of fully-formed classes of Comm and Listen. Feature refinements are often called layers.

For implementation we used Jak, a superset of Java for feature-oriented programming. The tool chain we used is based on Eclipse as a tool platform and FeatureIDE. FeatureIDE is a plug-in for Eclipse IDE and used to support the complete product line development process based on FOP. Figure 4.4 shows our equation file for Communication SPL. We can select features from the list and generate needed programs. UDP is a mandatory feature, so all programs generated from this product line have the communication functionality using UDP protocol over Ethenet. Encryption and Compression are optional features, we can select feature Encryption to generate a program which provides

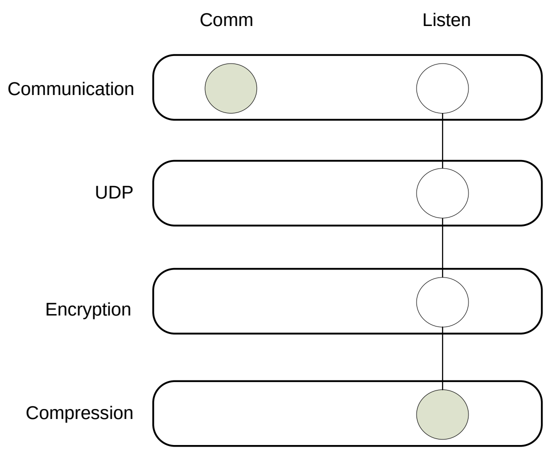
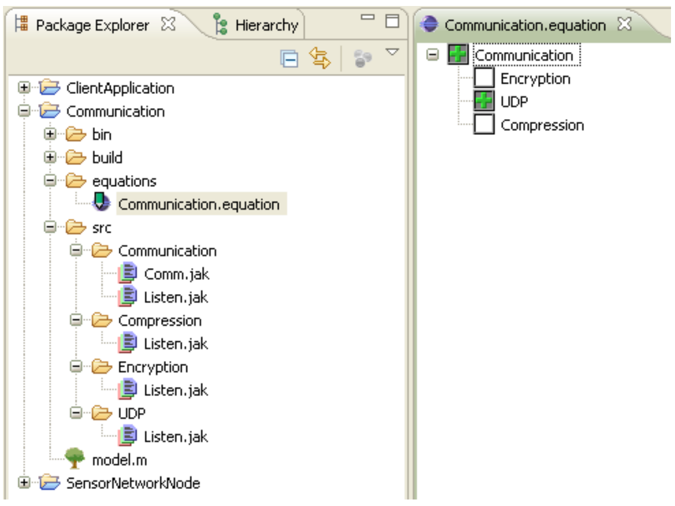


Figure 4.3: Classes and Refinements (Layers)

the encryption functionality during communication or encrypted data storage. We can also select feature Compression to generate a program which provides compression functionality or data storage. Furthermore, Encryption and Compression can be selected at the same time.



```
ClientApplication
Communication
  bin
  build
  equations
    Communication.equation
  src
    Communication
      Comm.jak
      Listen.jak
    Compression
      Listen.jak
    Encryption
      Listen.jak
    UDP
      Listen.jak
  model.m
SensorNetworkNode
```

```
Communication
  Encryption
  UDP
  Compression
```

Figure 4.4: The Equation File for Communication Product Line

4.1.3 Implementation of Dependent Software Product Lines

SensorNetworkNode SPL is used for developing software programs for sensor network nodes, and ClientApplication SPL is used to develop programs for different client hardware interfacing with sensor network nodes (e.g., Laptops and PDAs). We described the implementation of Communication SPL in the previous section. Then we will focus on the implementation of SensorNetworkNode and ClientApplication SPLs using Communication SPL to fulfill special functionalities. SensorNetworkNode product line may also use FAME-DBMS product line to implement data storage functionality.

Step-wise refinement is used to develop a complex and distributed software system from a simple software program by adding features incrementally. We implemented features NutOS, TinyOS, and Communication in SensorNetworkNode product line. We defined a Main Class in the SensorNetworkNode layer, then refinement NutOS and TinyOS refine this Main Class and modify this Main Class to realize special functionalities. All possible features for different client applications have been analyzed in domain analysis and described in a feature diagram. PDA and Laptop implement the functionality of screen output, whether the hardware interfacing with sensor network nodes is PDA or Laptop. We also defined a Main Class in ClientApplication layer. Then this Class is refined in PDA layer and Laptop layer to achieve their respective functions. The equation files for SensorNetworkNode and ClientApplication product lines are similar as the Communication SPL. Features can be arbitrary selected in the equation file to generate customized programs for different sensor nodes and clients.

Unfolding Feature Model References We can use the approach of unfolding feature model references to implement dependent SPLs. Specialization step allows us to unfold a feature model reference [CHE04]. We substitute the reference for the entire feature diagram it refers to by means of its root feature. This operation never removes variability. Feature Communication in SensorNetworkNode SPL and ClientApplication SPL requires functionalities provided by the Communication SPL. We unfold these feature model references as shown in Figure 4.5.

As described in the implementation of Communication SPL, Basic implementation is located in module Communication which is extended by feature modules UDP, Encryption and Compression that implement communication using UDP protocol over Ethenet, encryption and compression communication. If we want to implement the functionality of communication using special functionalities provided by Communication SPL, we need a variant of the Communication SPL. The idea is to define the concrete instance at deployment time and develop the SensorNetworkNode product line without creating a concrete instance. We can generate a specialized variant using staged configuration by removing features that should not be used. A concrete instance of the Communication product line has to be created when creating an instance of the SensorNetworkNode product line because the configuration is not known before.

Figure 4.6 depicts a specialized variant of Communication SPL by removing Feature Compression. This variant implements the functionality of encryption during communication using UDP protocol over Ethenet, and can be reused in SensorNetworkNode SPL.

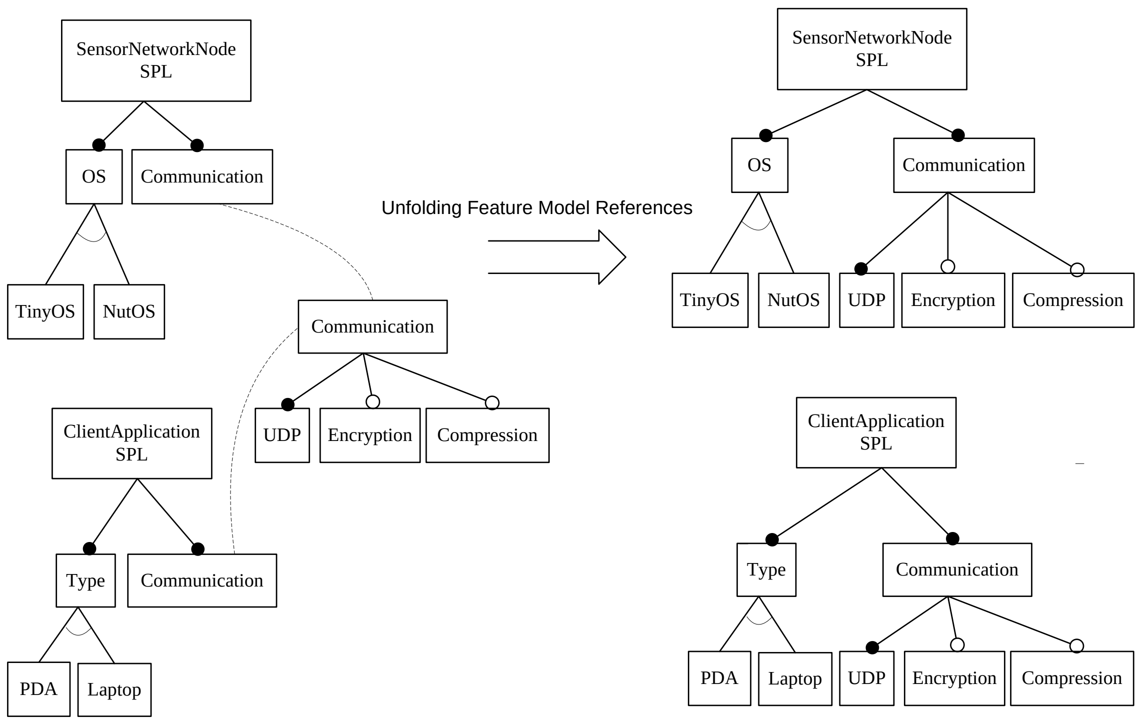


Figure 4.5: Unfolding Feature Model References

Similarly, if we want to implement the compression functionality, we have to generate a Compression variant at deployment time using staged configuration by removing Feature Encryption. In this example, we can generate four specialized variants of Communication product line:

- Communication using UDP protocol over Ethenet.
- Encrypted communication using UDP protocol over Ethenet.
- Compressed communication using UDP protocol over Ethenet.
- Encrypted and compressed communication using UDP protocol over Ethenet.

We implemented code reuse in dependent SPLs manually, but the source code copied from Communication SPL to SensorNetworkNode SPL must be composed Jak code. That means, we have to use the Communication SPL to produce the needed program, and reuse this program in SensorNetworkNode SPL. Furthermore, if we want to implement the functionality of data storage in sensor nodes, we have to generate concrete product from FAME-DBMS SPL and reuse it in SensorNetworkNode SPL. For example, a specialized variant of FAME-DBMS product line SPL is needed when storing data in

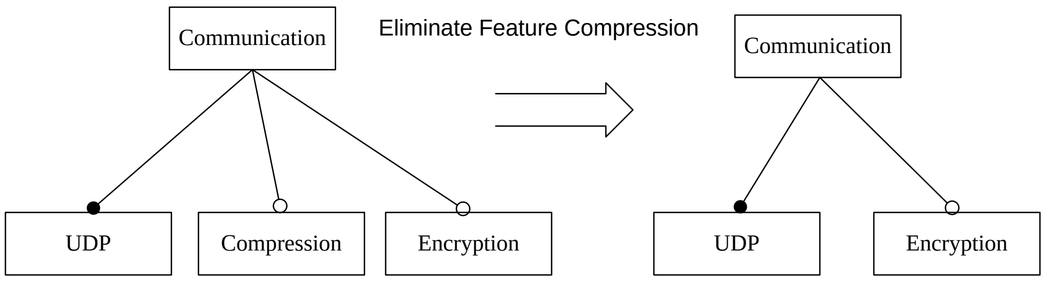


Figure 4.6: Staged Configuration of Communication Product Line by removing Feature Compression

sensor networks. We should define the concrete instance of FAME-DBMS which provides the functionality of data storage at deployment time. The approach of generating specialized variant of FAME-DBMS is similar as described above, using staged configuration by removing unneeded features. A concrete instance of the FAME-DBMS SPL has to be created when creating a SensorNetworkNode SPL instance.

Instances Constraints We can reference different instances of the same SPL in source code. This means that we want to instantiate different products from same SPL with differing functionalities. For example, a sensor node uses two different instances of an Encryption SPL, one used for encrypted data storage and another one for encryption communication. Or two different instances of Communication product line can be instantiated, one instance with encryption functionality, and the other without this functionality. The idea is to instantiate them separately at deployment time. Same as in Object-Oriented Programming we can have two different derived calls from same base class. Then we have to make two separate instances of derived classes. At instance level there are no dependencies of implementation if both are instances of same SPL. Dependencies arise when we instantiate products from two dependent SPLs. So if we want to reuse different products provided by the underlying SPL, we can reuse one instance of the underlying SPL which includes all required features. For example, we can reuse one instance of Communication product line which includes all required feature, such as UDP, Encryption, and Compression. But if one sensor node uses AES and the other uses DES, which cannot be configured in one instance. In this case, we need two different instances to provide different requirements for same SPL.

Interactions between multiple Features Crosscutting concerns are aspects of a program which affect other concerns and can be modularized using language like FeatureC++. However, interactions between multiple crosscutting features have to be considered [RSSA08]. For example, if we enable Encryption in FAME-DBMS we need a password. This password has to be provided by SensorNetworkNode SPL, but the password has to be provided only if the feature Encryption is used. We have two possibilities to implement code that is needed if a feature of a referenced product line is active.

- One possibility is to use constraints. We can add directly related feature below each of crosscut features. For example, we can create a feature Encryption in our sensor network node applications and a constraint that requires that feature if feature encryption in FAME-DBMS is seleced. The constraint is: `FAME-DBMS.Encryption` $\Rightarrow$ `SensorNetworkNode.Encryption`.

- The other possibility is to use derivatives. Derivatives can be modularized and separated from interacting features and are not different from other features but is only needed if the interacting features are present in an SPL instance. A derivative Fame.Encryption/Sensor.Encryption implements the code for providing the password.

The implementation is that we have to put the dependent code in a separate folder and create the shortcut/link to that folder with the name of feature for which we want to add this code, e.g., the code for using password. That means, we will put the code for using password as separate feature derivative in a folder. Then we will create the shortcut for this folder and will rename the shortcut with feature name i.e., encryption in this case. And put this shortcut in the folder of SensorNetworkNode SPL.

The solution using derivatives is better because the derivative is implicitly selected and we do not have to add a feature to the feature model of our SensorNetworkNode SPL. Based on implementation, derivatives are stored in a separate folder usually in a special location used only for derivatives. The symbolic links within folders of the corresponding features are used to ease navigation[1].

As described above, we used the idea of unfolding feature model references to implement our SensorNetworkNode SPL and ClientApplication SPL. Unfortunately, FeatureIDE doesn't support project references, so we have to copy the composed Jak code from Communication SPL to SensorNetworkNode SPL and ClientApplication SPL manually. Different variants of Communication SPL are generated and can be reused in other SPLs, this is working fine in our implementation. In the next section, we will focus on the configuration of dependent SPLs, and find a way to automate the configuration process.

[1]http://wwwiti.cs.uni-magdeburg.de/iti-db/forschung/fop/featurec/derivatives.htm

4.2 Configuring Dependent Software Product Lines

We can use the configuration tool to ensure that no configuration constraints are violated [CW04]. There are a few existing tools that support feature-based configuration: ConfigEditor [CBUE02], CaptainFeature [Bed02], GEARS [Kru01], Pure::Consul [VMP03]. The configuration of dependent SPLs is based on an instance model and domain models of all involved product lines, and should be supported by a configuration tool, and all of the underlying SPLs should not be visible to users. When creating a configuration, the user should start with an empty feature selection of the top level feature model. For example, the feature model of SensorNetwork SPL is the top level feature model in our sensor network scenario. This feature selection is based on the domain model of an SPL. Additional constraints in the instance model have to be checked during configuration. Dependent SPLs are only added when they are needed. After finishing the configuration of the SPL shown to the user, the configuration of the underlying SPLs follows [RSKuR08].

Then we will describe the configuration process of SensorNetwork SPL. We start with an empty feature selection which includes instances of SensorNode SPL and AccessNode SPL which are needed in sensor networks. If the user selects feature DataStorage in sensor network feature model, then an instance of DataNode SPL will be added according to conditional dependence `SensorNetwork.DataStorage` ⇒ `SensorNetwork.dataNode`, additionally instances of FAME-DBMS and DataNodeComm product lines will be added. Selection of feature Laptop adds an instance of Laptop SPL. After the selection of functional features, constraints have to be checked during configuration. Not covered constraints will be ignored. If feature PDA is selected, according to domain constraints `SensorNetwork.PDA` ⇒ `AccessNode.Bluetooth` feature Bluetooth is enabled in all instances of DataNode product line. After that, all dependent SPLs have to be configured. For example, if we selected accessNode and sensorNode in our configuration, then SensorNode SPL and AccessNode SPL have to be configured separately. For example, if feature AES is selected, instances of SensorNodeComm and AccessNodeComm will be added according to conditional dependencies `SensorNode.AES` ⇒ `SensorNode.sensorcom` and `AccessNode.AES` ⇒ `AccessNode.accesscom`. After all involved SPLs configured, our configuration process of dependent SPLs is finished. Our configuration process based on domain models and additionally checked against constraints of an instance model can be used to create a configuration generator for a composition of dependent SPLs, to achieve an automatic configuration.

Marko et al. developed an infrastructure to ease SPL instantiation and validation of instances in [RSSA08], i.e., SPL-API, which allows us to dynamically create SPL instances and avoid invalid configurations before instantiation. Client applications can access a product line model (PLM) which is stored in an XML files. We can access referenced feature models using functions provided by the class PLM such as PLM::Open and PLM::CreateInstance. We can open a used feature model from an XLM file and create instance according to a configuration provided as a list of features. The corresponding SPL can be instantiated and additionally checked against domain constraints for example. The SPL-API is implemented as a library and can be automatically bound to an application. According to the feature list in the XML files, a configuration can be generated automatically at a client development time [RSSA08]. We adopt this idea to achieve our automatic configuration process. The automatic configuration of dependent SPLs can be done with the help of SPL-API through storing a feature list in an XML file. We might extend our FeatureIDE with tools like SPL-API to automate the configuration of dependent SPLs.

4.3 Summary

In this chapter we described the implementation of Communication SPL, and the implementation of SensorNetworkNode and ClientApplication SPLs which use Communication and FAME-DBMS SPL to achieve special functionalities. In sensor network scenario, SensorNetwork SPL is integrated by multiple SPLs, i.e., SensorNetowrkNode, ClientApplication, Communication, and FAME-DBMS. In our case study, SensorNetwork SPL contains no source code. All codes are contained in the underlying SPLs. Ideally SPLs used within other SPLs should be automatically configured. Only functionalities of SensorNetwork SPL a user is interested in have to be configured manually. Automation is a benefit of implementing dependent SPLs and has to be done in further work using configuration generators. The source code of our case study is in the appendix.

Chapter 5

Evaluation

In this chapter we will evaluate the approaches of modeling and implementation presented of dependent SPLs in this thesis. The modeling approaches of dependent SPLs are presented in chapter three, i.e., feature model references and modeling SPL instances. The implementation approach of dependent SPLs is presented in chapter four. In the following, we will describe comparisons of the two modeling approaches. Drawbacks of feature model references and benefits of modeling SPL instances will be analyzed. However, there are also some issues of modeling SPL instances that have to be analyzed. Encountered problems when implementing dependent SPLs will be described. The evaluation is based on an existing FAME-DBMS SPL and additionally smaller SPLs that simulate a sensor network.

5.1 Evaluation of Modeling Approaches

Based on FODA, SPLs are usually described using feature models and domain models. Feature diagram is an appropriate approach to describe features within an SPL. Feature model references can be used to handle dependencies and constraints in the composition of multiple dependent SPLs. We use requires relation to describe dependencies between different product lines, and underlying SPLs have to be configured according to feature selection in the higher-level product line.

Although we can use feature model references to describe dependencies and constraints between different SPLs, but the drawback is that we cannot take concrete SPL instances into account using domain constraints, e.g., one SPL uses multiple differently configured instances of another SPL, and different configured instances of the same SPL are dependent on each other. In sensor networks, a sensor node may use two different instances of an encryption SPL, one used for encrypted data storage, and another one used for communication encryption. Differently configured instances of encryption SPL are the same product line (i.e., Communication SPL) using feature model references.

A sensor node may need the functionality of data storage node to store data. In this case, differently configured instances of one SPL (i.e., SensorNetworkNode SPL) are dependent on each other. We cannot describe dependencies and constraints using feature model references, which is the same for all nodes. We would refer to the same feature model, and not the concrete SPL instances.

Through the above examples, it is not difficult to see that the approach of using feature model references is not sufficient to describe compositions of dependent SPLs where differently configured instances of the same SPL are used. To solve this problem, we use a model that describes SPL instantiation [RSKuR08], and describe dependencies and constraints between differently configured SPL instances. As described in chapter three, we can not only define domain constraints and instance constraints in our model, but also conditional dependencies. The constraints presented in chapter three are only requires or implies constraints, but arbitrary propositional formulas might be supported. We can use Object Constraint Language (OCL) to specify constraints. There are also approaches to define constraints in domain models. Don Batory presented a fundamental connection between feature diagrams, grammars, and propositional formulas in [Bat05]. This connection enables efficient logic truth maintenance systems (LTMSs) which provide a way to propagate constraints as users select features, so that we can avoid inconsistent product specifications. This connection allows us to use satisfiability solvers which provide automated support to help debug feature models. D. Streitferdt et al. propose a formalized definition for feature modeling using OCL and a set of constraints to be used in feature models [SRP03]. Compared with domain models, all constraints have to define the product line name and the product line instance name in instance model. The approach of modeling SPL instances is an extension of the existing SPL modeling techniques.

Feature model references and modeling SPL instances are two useful approaches for modeling the composition of dependent SPLs. The approach of modeling SPL instances adopts the concept of class instantiation and uses class diagrams to model SPL compositions. SPL specialization is represented by inheritance and the uses relationship by aggregation. The approach of feature model references is independent of inheritance or aggregation, and allows us to reuse or modularize feature models. Feature model references do not consider relationships between features such as consists-of or is generalization-of which are better modeled using class diagrams in the approach of modeling SPL instances.

A feature model reference is represented using a dashed line. So if we want to modularize a large feature model over multiple different feature diagrams, a convoluted diagram cannot be avoid using feature model references. In contrast to feature model references, by using class diagrams complex compositions of dependent product lines can be created using existing tools.

Configuration of an SPL which is composed from multiple SPLs, all involved domain models and instance models has to be considered. When configuring SensorNetwork SPL, the user starts with an empty feature selection of the SensorNetwork prodcut line and selects functional features from the domain model. The underlying SPLs need not to be shown to the user, which should be supported by configuration tools. The feature selection of SensorNetwork SPL is based on the domain model of sensor networks, and we still have to check against constraints in the instance model. After finishing the configuration of one SPL, configuration of dependent SPLs follows. We hope that, there is no further feature selection of dependent SPLs needed, because we need to resolve constraints to support the automation of the configuration process of dependent SPLs. We should develop a configuration generator in the further work to achieve the automation of the configuration process of dependent SPLs. Existing support for generation of object-oriented from class diagrams can be used to derive configuration generator from SPL composition models.

Table 5.1: Comparison of Feature Model References and Modeling SPL Instances

	Feature Model References	Modeling SPL Instances
Constraints	Domain Constraints	Domain Constraints, Instance Constraints, Conditional Dependencies
Composition	Modularize Features	Using Class Diagrams
Specialization	Hierarchical Decomposition	Inheritance
Uses Relationship	Dashed Line	Aggregation
Configuration	exist invalid configurations	avoid invalid configurations

The comparison of feature model references and modeling SPL instances is shown in Table 5.1. As discussed above, it is complicated to use feature model references to describe dependencies and constraints between different SPLs and concrete SPL instances. Compared with feature model references, modeling SPL instances is a better approach to describe compositions of dependent SPLs as well as constraints between different concrete SPL instances. Feature model references can be used to describe compositions of different product lines only if an SPL uses one instance of the other SPLs. Modeling SPL instances can bridge the gap between domain and implementation. In domain analysis, we can use feature model reference to describe the dependencies and constraints on a domain model level. We can ease the transition from feature model to instance model using a model transformation. A class can be created for each feature model of a composition of different product lines. We can transform staged configuration of product lines into a hierarchy of classes to represent different configurations of product lines. Resulting classes or specialized variants of a product line can be linked to their feature model to support visualization [RSKuR08].

5.2 Evaluation of Implementation Approaches

We can derive a concrete product by selecting the desired features from an SPL. The corresponding feature modules are composed which results in an SPL instance. A program is created by composing class fragments of all classes according to the selected features. The concrete composition mechanism depends on the implementation technique. We can use FeatureC++ and Jak which implements AHEAD for C++ and Java to develop product lines. In this thesis we focused on the implementation of dependent SPLs using Jak and FeatureIDE. We created three feature projects (i.e., SensorNetworkNode, ClientApplication, and Communication) which represent three SPLs. SensorNetworkNode SPL and ClientApplication SPL use functionalities provided by Communication SPL, e.g., communication using UDP protocol, encryption, and compression. With the help of feature model references, we can unfold a feature reference, and define constraints between these features. For example, if we want to implement an encrypted communication between sensor nodes and client applications, we might create a feature Encryption both in SensorNetworkNode SPL and ClientApplication SPL. In our implementation, we first build our Communication project to generate a composed .jak file, and reuse this .jak file in SensorNetworkNode project and ClientApplication project. Because FeatureIDE doesn't support project reference, we cannot directly call functions belongs to other projects. An error will occur when compiling Jak code using project reference. This problem of FeatureIDE should be further analyzed in the future. Another problem we encountered is that we cannot use package in our implementation, otherwise, an error will occur.

Based on the implementation of sensor networks described in the previous chapter, we can create four instances of Communication SPL. If we use instance constraints to implement the code reuse in other product line, we have to define different instance names. The SPL-API developed by Marko et al. in [RSSA08] is helpful to ease SPL instantiation and validation of instances. It allows us to dynamically create SPL instances and validate configurations before instantiation. In our implementation we discovered that a correct ordering of features is needed to create a semantically correct instance of an SPL. We can create correct ordering of features according to the relative ordering of features at runtime using SPL-API. We have two approaches to solve interactions between multiple features, one possibility is to use constraints, and the other is to use derivatives. The solution using derivatives is better because the derivative is implicitly selected and we do not have to add a feature to the feature model. We also can further decrease the complexity of SPL composition at runtime using derivatives. The SPL-API can apply a derivative in the dynamic composition process, if the features which this derivative belongs are presented in the configuration process.

Because implementation components can only be connected in certain ways, we can describe their valid configurations using languages, e.g., DSL. A configuration SPL allows us to specify a concrete instance of a product line. All instances can be correctly described by the modeling approaches such as modeling SPL instances. Feature model references allow invalid instances and can not describe all instances. In the configuration process of our implementation, we only want to configure the SensorNetworkNode SPL and ClientApplication SPL. The underlying Communication SPL should be configured automatically according to the feature selection in SensorNetworNode SPL and ClientApplication SPL. We cannot automate this configuration process using FeatureIDE because a configuration generator is missing.

Chapter 6

Conclusion

In this thesis, we use a case study sensor network to describe the modeling and implementation of dependent SPLs. Feature model references and modeling SPL instances are two appropriate approaches to implement compositions of multiple dependent SPLs.

SPLs can be reused as part of other product lines. Sometimes, functionalities and components of multiple product lines have to be integrated into one larger product line [Omm02], e.g., SensorNetwork SPL. Development of complex and distributed software programs, we may have multiple SPLs, usually dependencies between them exist. Compositions of multiple SPLs imply that we have to consider about the dependencies between those SPLs. Feature model references can be used to handle dependencies and constraints between different product lines. In this case, we can use domain constraints to describe dependencies between different SPLs. Requires relations can be used to describe dependencies and constraints between different product lines.

Although we can use domain constraints to describe dependencies and constraints between different product lines, but the drawback is that we cannot take concrete instances of product line into account using domain constraints. We cannot describe dependencies and constraints, when one product line uses multiple differently configured instances of another product line, or when different configured instances of the same product line are dependent on each other. The approach of using feature model references for modeling dependencies between different SPLs is not sufficient to describe compositions of dependent product lines where differently configured instances of the same product line are used. To solve this problem, we can use the approach of modeling SPL instances that describes SPL instantiation, and describe dependencies and constraints between differently configured SPL instances. We can not only define domain constraints and instance constraints in our model, but also conditional dependencies. The constraints presented in chapter three are only requires or implies constraints, but arbitrary propositional formulas might be supported.

We described the implementation of an SPL and dependent SPLs. The tool chain we used in our case study is based on Eclipse as a tool platform. Tool Support for Feature-Oriented Software Development is FeatureIDE (a plug-in for the Eclipse IDE). With the help of feature model references and modeling SPL instances we implemented a real sensor network system.

We summarize various conclusions in the light of goals we established at the beginning of the thesis:

- We compared and evaluated the approaches for compositions of dependent SPLs. Feature model references and modeling SPL instances are two appropriate approaches for compositions of dependent SPLs. But when taking multiple SPL instances into account, feature model references are not sufficient to describe dependencies and constraints between SPLs and SPL instances.

- With respect of assets reuse, we analyzed all possible features of a communication SPL which provides functionalities for sensor network node product line and client application product line in our case study, and is used for evaluation of the approaches of feature model reference and modeling SPL instances.

- Based on existing FAME-DBMS we developed smaller SPLs such as SensorNetworkNode, ClientApplication, and Communication product lines to simulate a real sensor network system. With the help of unfolding feature model references we implemented a sensor network system which provides functionalities of UDP communication protocol, encryption, and compression. Modeling SPL instances are helpful for referencing different instances of the same SPL in source code.

6.1 Further Work

In this thesis, the modeling and implementation approaches of dependent SPLs have been presented, but we still have a lot of work of modeling and implementation of dependent SPLs to be completed in the future.

Automatic Configuration The build process can be automated with tools like Make, we also want to automate configuration process and quality control activities. Development of a configuration generator can realize automatic configuration of dependent SPLs. We can use the model presented in this thesis as the basis for a configuration generator with the help of resolving constraints. Using configuration generator a user only need to select functional features in the top level SPL, the underlying SPLs will be configured automatically according to constraints defined between dependent SPLs.

Tool Support Through our case study, we find that the lack of tool support is a big problem in the development of sensor networks. We used FeatureIDE as a development platform, but it doesn't support project reference when compiling Jak code which uses functions belongs to other projects. Package is also not allowed to be used. In order to better make use FeatureIDE to realize the implementation of dependent SPLs, we have to expand FeatureIDE to solve the problems we encountered. In the further work, we want to develop an automated configuration process as part of FeatureIDE through the use of domain constraints and instance constraints.

Bibliography

[Bat05] Batory, D.: *Feature Models, Grammars, and Propositional Formulas. In Proceedings of the International Software Product Line Conference (SPLC), Volume 3714 of Lecture Notes in Computer Science, Pages 7-20.* Springer Verlag, 2005.

[Bed02] Bednasch, T.: *Konzept und Implementierung eines konfigurierbaren Metamodells für die Merkmalmodellierung. Diplomarbeit.* Fachhochschule Kaiserslautern,Standort Zweibrücke Fachbereich Informatik, 2002.

[BSR04] Batory, D.; Sarvela, J. N.; Rauschmayer, A.: *Scaling Step-Wise Refinement IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 30, NO. 6.* 06 2004.

[CBUE02] Czarnecki, K.; Bednasch, T.; Unger, P.; Eisenecker, U.: *Generative Programming for Embedded Software: An Industrial Experience Report. In Proceedings of the ACM.* SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE02), Pittsburgh, 2002.

[CE00] Czarnecki, K.; Eisenecker, U. W.: *Generative Programming—Methods,Tools,and Application.* Addison-Wesley, Canada, 2000.

[CHE04] Czarnecki, K.; Helsen1, S.; Eisenecker2, U.: *Staged Configuration Using Feature Models.* Spring Verlag, University of Waterloo, Canada and University of Applied Sciences Kaiserslautern, Zweibruecken, Germany, 2004.

[CHE05a] Czarnecki, K.; Helsen1, S.; Eisenecker2, U.: *Formalizing Cardinality-based Feature Models and their Staged Configuration.* University of Waterloo, Canada and University of Applied Sciences Kaiserslautern, Zweibruecken, Germany, 2005.

[CHE05b] Czarnecki, K.; Helsen1, S.; Eisenecker2, U.: *Staged Configuration Through Specialization and Multi-Level Configuration of Feature Models.* University of Waterloo, Canada and University of Applied Sciences Kaiserslautern, Zweibruecken, Germany, 2005.

[CW04] Czarnecki, K.; Weiland, J.: *Variant Configuration of Software Systems. Process Family Engineering in Service-Oriented Applications, BMBF-Project.* 2004.

[Kru01] Krueger, C. W.: *Software Mass Customization.* White paper, BigLever Software Inc., Octoer 2001.

[LAM05] Leich, T.; Apel, S.; Marnitz, L.: *Tool Support for FeatureOriented Software Development,FeatureIDE: An EclipseBased Approach.* Nr. MIP-0802. San Diego, California, 2005.

[Mah07] Mahalik, N. P.: *Sensor Networks and Configuration. Fundamentals, Standards, Platforms, and Applications.* Springer Verlag, 2007.

[Omm02] Ommering, R. V.: *Building Product Populations with Software Components. In Proceedings of the international Conference on Software Engineering (ICSE), pages 255-265.* ACM Press, 2002.

[PBvdL05] Pohl, K.; Boeckle, G.; Linden, F. v. d.: *Software Product Line Engineering.* Springer, Heidelberg, 2005.

[RSKuR08] Rosenmüller, M.; Siegmund, N.; Kästner, C.; Rahman, S. S. u.: *Modeling Dependent Software Product Lines.* Nr. MIP-0802. Department of Informatics and Mathematics, University of Passau, Nashville, TN, USA, 10 2008.

[RSS$^+$08] Rosenmüller, M.; Siegmund, N.; Schirmeier, H.; Sincero, J.; Apel, S.; Leich, T.; Spinczyk, O.; Saake, G.: *FAME-DBMS: Tailor-made Data Management Solutions for Embedded Systems.* ACM International Conference Proceeding Series, 2008.

[RSSA08] Rosenmüller, M.; Siegmund, N.; Saake, G.; Apel, S.: *Code Generation to Support Static and Dynamic Composition of Software Product Lines.ACM.* 2008.

[SMZ07] Sohraby, K.; Minoli, D.; Zanti, T.: *Wireless Sensor Networks—Technology, Protocols, and Applications.* Wiley Interscience, 2007.

[SRP03] Streitferdt, D.; Riebisch, M.; Philippow, I.: *Details of formalized relations in feature models using OCL. pages 297-304. IEEE Computer Society Press.* 2003.

[VMP03] *Variant Management with Pure::Consul. Technical White Paper.pure-systems.* http://web.pure-systems.com/, 2003.

[Zen07] Zenker, R.: *Bluetooth Sensor Networks.* http://www-comnet.technion.ac.il/ralf/Bluetooth-Sensor-Network.pdf, Technion, Israel Institute of Technology Communication Networks Lab, 2007.

Appendix

Source code of Communication SPL We have implemented Features UDP, Compression, and Encryption in Communication SPL. Each layer represents a feature in the source code.

```
layer Communication;

import java.io.*;
import java.net.*;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
public class Comm {
public Listen l;
public void verhaltung(InetAddress ip,String msg){
}
}

layer Communication;

import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.SocketException;
public class Listen extends Thread {
public int port;
public DatagramSocket dsocket;
public Comm comm;
public byte[] buffer = new byte[2048];
public DatagramPacket packet;
public long time;
public String msg;
public void send(InetAddress ip, int port, String msg) {}
}

layer Compression;

public refines class Listen {
```

```
public void send(InetAddress ip, int port, String msg) {
Super(InetAddress,int,String).send(ip,port,this.msg);
// compression method
}
public void verhaltung(InetAddress ip,String msg){
Super(InetAddress,String).verhaltung(ip,msg);
// compression method
}
}

layer Encryption;

public refines class Listen {
public void send(InetAddress ip, int port, String msg) {
this.msg=msg.substring(0,13)+rot13(msg.substring(13));
}
public void verhaltung(InetAddress ip,String msg){}
public String rot13(String pf) {
StringBuffer pf2 = new StringBuffer();
for (int i = 0; i < pf.length(); i++) {
int c = pf.charAt(i);
if (c < 65 || c > 122) {
pf2.append((char) c);
} else if ((c < 78 && c >= 65) || (c < 110 && c >= 97)) {
pf2.append((char) (c + 13));
} else if ((c >= 78 && c <= 90) || (c >= 110 && c <= 122)) {
pf2.append((char) (c - 13));
}
}
return pf2.toString();
}
}

layer UDP;

public refines class Listen {
public int port;
public DatagramSocket dsocket;
public Comm comm;
public byte[] buffer = new byte[2048];
public DatagramPacket packet;
public long time;
public String msg;
public Listen(int p, Comm comm) {
this.port = p;
try {
this.dsocket = new DatagramSocket(port);
packet = new DatagramPacket(buffer, buffer.length);
} catch (SocketException e) {

e.printStackTrace();
}
this.comm = comm;
time=System.nanoTime();
}
public void run() {
while (true) {
try {
dsocket.receive(packet);
} catch (IOException e) {
e.printStackTrace();
}
// Convert the contents to a string, and display them
String msg = new String(buffer, 0, packet.getLength());
System.out.println(packet.getAddress().getHostName() + ": " + msg);
verhaltung(packet.getAddress(),msg);
// Reset the length of the packet before reusing it.
packet.setLength(buffer.length);
}
}
public void send(InetAddress ip, int port, String msg) {
this.msg=msg;
Super(InetAddress,int,String).send(ip,port,this.msg);
long timet=System.nanoTime()-this.time;
this.time=System.nanoTime();
String msgt=this.msg+String.valueOf(timet);
byte[] message = msgt.getBytes();
DatagramPacket packet = new DatagramPacket(message, message.length, ip,
port);
try {
DatagramSocket dsocket = new DatagramSocket();
dsocket.send(packet);
dsocket.close();
} catch (Exception e) {
System.err.println(e);
}
}
public void verhaltung(InetAddress ip,String msg){
this.msg=msg;
comm.verhaltung(ip,this.msg);
}
}
```

Source code of SensorNetworkNode SPL We have implemented Features NutOS, TinyOS, and Communication in SensorNetworkNode SPL. Each layer represents a feature in the source code. Layer NewEquation means the code reuse of other SPLs.

```
layer SensorNetwork_Node_SPL;

public class Main {
public static void main(String[] args){
 new Main().PrintOS();
 //new Comm().communication();
}
public void PrintOS(){}
}

layer TinyOS;

public refines class Main {
public void PrintOS(){
System.out.println("TinyOS");
}
}

layer NutOS;

public refines class Main {
public void PrintOS(){
System.out.println("NutOS");
}
}

layer Communication;

public refines class Main {
public static void main(String[] args){
Super(String[]).main(args);
new Comm();
}
}

layer Communication;

import java.io.*;
import java.net.*;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
public class Comm {
public Listen l;
public Comm(){
System.out.println("Sensor");
l=new Listen(9001,this);
l.start();
try {
InetAddress ip=InetAddress.getByName("127.0.0.1");
String time=String.valueOf(System.nanoTime());
l.send(ip,9100,"time:"+time);
} catch (UnknownHostException e) {
e.printStackTrace();
}
}

public void verhaltung(InetAddress ip,String msg){
String temp=msg.substring(0,13);
String temp1=msg.substring(13);
if (temp.equals("SensorNetwork")){
l.send(ip,9300,temp+temp1.toUpperCase());
};
}
}

layer NewEquation;

import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.SocketException;

SoUrCe RooT
Communication "../../H:/workspace/Communication_spl/src/Communication/Listen.jak";

abstract class Listen$$Communication extends Thread {
public int port;
public DatagramSocket dsocket;
public Comm comm;
public byte[] buffer = new byte[2048];
public DatagramPacket packet;
public long time;
public String msg;
public void send(InetAddress ip, int port, String msg) {}
}

SoUrCe  Encryption "../../H:/workspace/Communication_spl/src/Encryption/Listen.jak";

abstract class Listen$$Encryption extends  Listen$$Communication {
```

```
public void send(InetAddress ip, int port, String msg) {
this.msg=msg.substring(0,13)+rot13(msg.substring(13));
}
public void verhaltung(InetAddress ip,String msg){}
public String rot13(String pf) {
StringBuffer pf2 = new StringBuffer();
for (int i = 0; i < pf.length(); i++) {
int c = pf.charAt(i);
if (c < 65 || c > 122) {
pf2.append((char) c);
} else if ((c < 78 && c >= 65) || (c < 110 && c >= 97)) {
pf2.append((char) (c + 13));
} else if ((c >= 78 && c <= 90) || (c >= 110 && c <= 122)) {
pf2.append((char) (c - 13));
}
}
return pf2.toString();
}
}

SoUrCe  UDP "../../H:/workspace/Communication_spl/src/UDP/Listen.jak";

public class Listen extends  Listen$$Encryption  {
public int port;
public DatagramSocket dsocket;
public Comm comm;
public byte[] buffer = new byte[2048];
public DatagramPacket packet;
public long time;
public String msg;
public Listen(int p, Comm comm) {
this.port = p;
try {
this.dsocket = new DatagramSocket(port);
packet = new DatagramPacket(buffer, buffer.length);
} catch (SocketException e) {
e.printStackTrace();
}
this.comm = comm;
time=System.nanoTime();
}
public void run() {
while (true) {
try {
dsocket.receive(packet);
} catch (IOException e) {
e.printStackTrace();
}
// Convert the contents to a string, and display them
String msg = new String(buffer, 0, packet.getLength());
System.out.println(packet.getAddress().getHostName() + ": " + msg);
verhaltung(packet.getAddress(),msg);
// Reset the length of the packet before reusing it.
packet.setLength(buffer.length);
}
}
public void send(InetAddress ip, int port, String msg) {
this.msg=msg;
Super(InetAddress,int,String).send(ip,port,this.msg);
long timet=System.nanoTime()-this.time;
this.time=System.nanoTime();
String msgt=this.msg+String.valueOf(timet);
byte[] message = msgt.getBytes();
DatagramPacket packet = new DatagramPacket(message, message.length, ip, port);
try {
DatagramSocket dsocket = new DatagramSocket();
dsocket.send(packet);
dsocket.close();
} catch (Exception e) {
System.err.println(e);
}
}
public void verhaltung(InetAddress ip,String msg){
this.msg=msg;
comm.verhaltung(ip,this.msg);
}
}
```

Source code of ClientApplication SPL We have implemented Features PDA, Laptop, and Communication in ClientApplication SPL. Each layer represents a feature in the source code. Layer NewEquation means the code reuse of other SPLs.

```
layer ClientApplication;

public class Main {
public static void main(String[] args){
 new Main().PrintOS();
}
public void PrintOS(){}
}

layer Laptop;
```

```
public refines class Main {
public void PrintOS(){
System.out.println("Laptop");
}
}

layer PDA;

public refines class Main {
public void PrintOS(){
System.out.println("PDA");
}
}

layer Communication;

public refines class Main {
public static void main(String[] args){
Super(String[]).main(args);
new Comm();
}
}

layer Communication;

import java.io.*;
import java.net.*;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
public class Comm {
public Listen l;
public void verhaltung(InetAddress ip,String msg){
l.send(ip,9001,"SensorNetwork00abcdefghijklmn00");
}
public Comm(){
System.out.println("Client");
l=new Listen(9300,this);
l.start();
try {
InetAddress ip=InetAddress.getByName("127.0.0.1");
l.send(ip,9001,"SensorNetwork00abcdefghijklmn00");
} catch (UnknownHostException e) {
e.printStackTrace();
}
}
}
```

```
layer NewEquation;

import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.SocketException;

SoUrCe RooT
Communication "../../H:/workspace/Communication_spl/src/Communication/Listen.jak";

abstract class Listen$$Communication extends Thread {
public int port;
public DatagramSocket dsocket;
public Comm comm;
public byte[] buffer = new byte[2048];
public DatagramPacket packet;
public long time;
public String msg;
public void send(InetAddress ip, int port, String msg) {}
}

SoUrCe  Encryption "../../H:/workspace/Communication_spl/src/Encryption/Listen.jak";

abstract class Listen$$Encryption extends  Listen$$Communication  {
public void send(InetAddress ip, int port, String msg) {
this.msg=msg.substring(0,13)+rot13(msg.substring(13));
}
public void verhaltung(InetAddress ip,String msg){}
public String rot13(String pf) {
StringBuffer pf2 = new StringBuffer();
for (int i = 0; i < pf.length(); i++) {
int c = pf.charAt(i);
if (c < 65 || c > 122) {
pf2.append((char) c);
} else if ((c < 78 && c >= 65) || (c < 110 && c >= 97)) {
pf2.append((char) (c + 13));
} else if ((c >= 78 && c <= 90) || (c >= 110 && c <= 122)) {
pf2.append((char) (c - 13));
}
}
return pf2.toString();
}
}

SoUrCe  UDP "../../H:/workspace/Communication_spl/src/UDP/Listen.jak";
```

```
public class Listen extends  Listen$$Encryption  {
public int port;
public DatagramSocket dsocket;
public Comm comm;
public byte[] buffer = new byte[2048];
public DatagramPacket packet;
public long time;
public String msg;
public Listen(int p, Comm comm) {
this.port = p;
try {
this.dsocket = new DatagramSocket(port);
packet = new DatagramPacket(buffer, buffer.length);
} catch (SocketException e) {

e.printStackTrace();
}
this.comm = comm;
time=System.nanoTime();
}
public void run() {
while (true) {
try {
dsocket.receive(packet);
} catch (IOException e) {
e.printStackTrace();
}
// Convert the contents to a string, and display them
String msg = new String(buffer, 0, packet.getLength());
System.out.println(packet.getAddress().getHostName() + ": " + msg);
verhaltung(packet.getAddress(),msg);
// Reset the length of the packet before reusing it.
packet.setLength(buffer.length);
}
}
public void send(InetAddress ip, int port, String msg) {
this.msg=msg;
Super(InetAddress,int,String).send(ip,port,this.msg);
long timet=System.nanoTime()-this.time;
this.time=System.nanoTime();
String msgt=this.msg+String.valueOf(timet);
byte[] message = msgt.getBytes();
DatagramPacket packet = new DatagramPacket(message, message.length, ip,
port);
try {
DatagramSocket dsocket = new DatagramSocket();
dsocket.send(packet);
dsocket.close();
} catch (Exception e) {
System.err.println(e);
}
}
public void verhaltung(InetAddress ip,String msg){
this.msg=msg;
comm.verhaltung(ip,this.msg);
}
}
```

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur mit erlaubten Hilfsmitteln angefertigt habe.

Magdeburg, den 2nd June 2009

Tao Wei