AN APPROACH TO MANAGE VARIABILITY IN OBJECT-ORIENTED APPLICATIONS WITH FEATURE MODELS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ENDER BULUT

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

SEPTEMBER 2014

Approval of the thesis:

AN APPROACH TO MANAGE VARIABILITY IN OBJECT-ORIENTED APPLICATIONS WITH FEATURE MODELS

submitted by **ENDER BULUT** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen _____
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Adnan Yazıcı _____
Head of Department, **Computer Engineering**

Dr. Cevat Şener _____
Supervisor, **Computer Engineering Department, METU**

Examining Committee Members:

Prof. Dr. Ali H. Doğru
Computer Engineering Department, METU _____

Dr. Cevat Şener
Computer Engineering Department, METU _____

Prof. Dr. Ahmet Coşar
Computer Engineering Department, METU _____

Assist. Prof. Dr. Selim Temizer
Computer Engineering Department, METU _____

Burak İbrahim Sevindi, M.Sc.
Computer Engineer, TÜBİTAK _____

Date: _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: ENDER BULUT

Signature :

ABSTRACT

AN APPROACH TO MANAGE VARIABILITY IN OBJECT-ORIENTED APPLICATIONS WITH FEATURE MODELS

Bulut, Ender
M.S., Department of Computer Engineering
Supervisor : Dr. Cevat Şener

September 2014, 66 pages

In this thesis, an approach to manage variability in object-oriented applications by using a feature modeling language and a simple source code generation technique has been designed and developed. This approach provides developing configurable object oriented applications in a practical way. That is, an application developed with our approach takes just a configuration file including user selections in a pre-defined domain as input and then automatically configure and manage itself with respect to these selections.

To build an application by using this approach, there are several steps that should be followed. First of all, the variability points in the domain are specified and designed by using feature model. After the feature model is designed, the code generator implemented for our approach produces automatically the semi-automatic source code templates containing interfaces, configurator classes and entity classes with attributes designed specially as data structures to be used to manage variability. Software developers who are experts in software architecture will be responsible for the next step. The developers manually develop the application with the generated source code templates by using the data structures considering the commonality and variability of the domain and so the application can be provided to work for various feature combinations. In the final step, a developer or even a customer that can understand the feature diagram selects the desired features of the application according to the rules and con-

straints in the configurator of a feature modeling language like FeatureIDE and so that the variabilities in the domain can be reflected automatically and directly to this object-oriented application with the feature model. Moreover, the proposed approach is applied to an online shopping domain as a case study in order to demonstrate concretely the contribution of this work.

Keywords: Variability Management, Code Generation, Object-Oriented Applications, Domain Analysis

ÖZ

NESNE YÖNELİMLİ UYGULAMALARDA DEĞİŞKENLİĞİ YETENEK MODELLERİ İLE YÖNETMEK İÇİN BİR YAKLAŞIM

Bulut, Ender
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Dr. Cevat Şener

Eylül 2014 , 66 sayfa

Bu tezde, nesneye dayalı uygulamalarda yetenek modeli ve basit bir kaynak kod üretme tekniği kullanılarak değişkenliği yönetmeye çalışan bir yaklaşım dizayn edildi ve geliştirilmiştir. Bu yaklaşım pratik bir yol ile nesneye dayalı konfigure edilebilir uygulamalar geliştirebilmeyi sağlar. Yani, bu yöntemle geliştirilen bir uygulama girdi olarak sadece önceden tanımlanmış olan bir alandaki kullanıcı seçimlerini içeren bir konfigurasyon dosyasını alır ve daha sonra, kendini bu seçimlere göre otomatik olarak yapılandırıp yönetir.

Bu yaklaşımı kullanarak bir uygulama geliştirmek için takip edilmesi gereken bir kaç adım vardır. Öncelikle, bir alandaki değişkenlik noktaları belirlenir ve yetenek modeli kullanılarak tasarlanır. Yetenek modeli tasarlandıktan sonra, bizim metodumuz için gerçekleştirilmiş olan kod üreticisi ara birimleri, konfigürasyon sınıflarını ve özel olarak değişkenliği yönetmek için kullanılmak üzere veri yapıları olarak tasarlanmış nitelikleri ile birlikte varlık sınıflarını içeren yarı otomatik kaynak kod taslaklarını otomatik olarak üretir. Yazılım mimarisinde uzman olan yazılım geliştiriciler de bir sonraki adımdan sorumlu olacaklardır. Bu geliştiriciler üretilmiş olan bu taslak kaynak kodlarını, uygulamanın ortaklığını ve değişkenliğini göz önünde bulunduran veri yapılarını kullanarak manuel olarak geliştirir ve böylece, bu uygulamanın değişik yetenek kombinasyonları için çalışması sağlanır. Son adımda, yetenek diyagramını anlayabilen bir geliştirici veya bir müşteri bile FeatureIDE'nin konfigürasyon tanım-

layıcısında kurallar ve zorunluluklara göre uygulamada olmasını istediği yetenekleri seçebilir ve böylece alandaki değişkenlikler doğrudan doğruya ve otomatik olarak yetenek modeli aracılığıyla bu nesneye dayalı uygulamaya yansıtılabilir. Ayrıca bu çalışmanın katkısını daha iyi gösterebilmek için, bu önerilen yaklaşım örnek bir olay incelemesi olarak internet üzerinden alışveriş alanına uygulanmıştır.

Anahtar Kelimeler: Değişkenlik Yönetemi, Kod Üretimi, Nesneye Dayalı Uygulamalar, Alan Analizi

To my family and Tuğba

ACKNOWLEDGMENTS

First of all, I would like to thank my supervisor Dr. Cevat Şener for his real support, guidance and friendship. He motivated and communicated with me highly all the time. It was a great honor for me to work with him for the last two years. I would like to thank also Professor Ali H. Doğru and Associate Professor Selim Temizer for their comments and guidance about my thesis.

A lot of people are with me and they support me in my all education life. They matured me and made me who I am during this difficult but beautiful life. They are real owners of this study. Above all, first and foremost they loved me. It is difficult to separately write here the importance of them. I am very grateful to all these people; they changed me deeply in terms of philosophy of my life, my vision, happiness and friendship. I feel very lucky to have them all. So I want to write the names of some of them; my best friends Burak Kerim Akkuş, Gökhan Cansız, my friends from work, Burak İbrahim Sevindi, Seyfullah Demir and Ahmet Emre Kılınç.

I would also like to thank you Tuğba Okutan for always listening me and making me happy with her sweetness and beauty through all difficult times. I love her so much and I want to be with her all my life because I really know that she always supports me and makes me strong.

Lastly, I want to write a separate paragraph to my family members supporting me for my each decision. Sincerest thanks to my mother who actually deserves all congratulations for success of me does for me whole my life, my father who trusts me about everything and my brother who loves me so much and makes all my demands without questioning. Each of them has taught me that the most important thing in this life is family and love.

TABLE OF CONTENTS

ABSTRACT v

ÖZ vii

ACKNOWLEDGMENTS x

TABLE OF CONTENTS xi

LIST OF FIGURES xiv

LIST OF ABBREVIATIONS xvi

CHAPTERS

1 INTRODUCTION 1

2 BACKGROUND 5

2.1 Feature 5

2.2 Feature-oriented Software Development (FOSD) 6

2.3 Domain Engineering 8

2.4 Feature Modeling 9

2.4.1 Basic Feature Models 9

2.4.2 Extended Feature Models 10

2.5 Feature Diagram 11

2.6 Configuration 13

2.7 Software Commonality and Variability 13

2.8 Variability Management 14

3 RELATED WORK 15

4 MANAGING APPLICATION VARIABILITY 23

4.1 Feature Modeling Language: FeatureIDE 25

4.2 Basic Assumptions in Our Approach 28

4.3 Recursive Parsing of Feature Model 29

4.4 Generating Source Code Templates with Data Structures . . . 30

4.4.1 Entity Classes and Interfaces with The Content of Data Structures 31

4.4.2 *Mandatory* Connection 31

4.4.3 *Optional* Connection 33

4.4.4 *And* Connection 34

4.4.5 *Alternative* Connection 35

4.4.6 *Or* Connection 37

4.4.7 Abstract Feature 38

4.4.8 Configuration Classes 41

5 CASE STUDIES IN ONLINE SHOPPING DOMAIN 45

5.1 A Synthetic Model 45

5.2 Online Shopping Domain 48

5.2.1 Feature Model for E-Shop System 49

5.2.2 UML Class Diagram for E-Shop Feature Model . . 52

5.2.3 Generation of Class Templates 53

5.2.4 E-Shop Application 55

5.2.5 Possible Combinations of E-Shop Feature Model . 55

6 CONCLUSION 59

6.1 Summary of Conducted Approach 59

6.2 Discussion 60

6.3 Future Work 61

REFERENCES 63

LIST OF FIGURES

FIGURES

Figure 2.1 Domain Engineering and Application Engineering (adapted from [2]) 8

Figure 2.2 A basic feature model sample (adapted from [11]) 11

Figure 2.3 Feature Relationships in Feature Diagram (adapted from [8]) 12

Figure 2.4 Feature Diagram Sample in FeatureIDE Tool 12

Figure 2.5 A configuration sample in FeatureIDE tool 13

Figure 3.1 An Example Workflow of AHEAD Toolkit (adapted from [9]) . . . 19

Figure 3.2 The Workflow of Rhizome Platform (adapted from[18]) 20

Figure 3.3 The Workflow of FORM Extension Method (adapted from [24]) . . 21

Figure 4.1 The Workflow of Our Approach 24

Figure 4.2 A Sample Feature Diagram in FeatureIDE 26

Figure 4.3 The XML Notation of An Example Feature Diagram in FeatureIDE 27

Figure 4.4 An Invalid (a) and A Valid (b) Feature Configuration in FeatureIDE 28

Figure 4.5 A Mandatory Feature (a) and The Corresponding UML Class Diagram (b) 32

Figure 4.6 Generated Source Code for Mandatory Connection 32

Figure 4.7 An Optional Feature (a) and The Corresponding UML Class Diagram (b) 33

Figure 4.8 The *And* Feature (a) and The Corresponding UML Class Diagram (b) 34

Figure 4.9 The *Alternative* Features (a) and The Corresponding UML Class Diagram (b) 35

Figure 4.10 Generated Source Code for *Alternative* Connection 36

Figure 4.11 *Or* Connection Features (a) and The Corresponding UML Class Diagram (b) 37

Figure 4.12 Generated Source Code for *Or* Connection 39

Figure 4.13 *Abstract* Feature (a) and The Corresponding UML Class Diagram (b) 40

Figure 4.14 An Example Feature Model 41

Figure 4.15 An Example UML Class Diagram 43

Figure 5.1 A Synthetic Model in FeatureIDE 46

Figure 5.2 UML Class Diagram for The Synthetic Model 47

Figure 5.3 Online Shopping Principle 48

Figure 5.4 E-Shop Feature Model 50

Figure 5.5 UML Class Diagram of E-Shop System 53

Figure 5.6 An Invalid Configuration of E-Shop System 56

Figure 5.7 An Invalid Configuration of E-Shop System 57

LIST OF ABBREVIATIONS

COSE	Component-Oriented Software Engineering
COSEML	Component-Oriented Software Engineering Feature Modeling
FODA	Feature-Oriented Domain Analysis
FOP	Feature-Oriented Programming
FORM	Feature-Oriented Reuse Method
FOSD	Feature-Oriented Software Development
OOP	Object-Oriented Programming
OVM	Orthogonal variability Modeling
SPL	Software Product Line
SPLE	Software Product Line Engineering
UML	Unified Modeling Language
XML	Extensible Markup Language

CHAPTER 1

INTRODUCTION

In recent years, software companies has been searching new methods and techniques in order to practically produce more user friendly and high quality software products in a short time. The common solution for these problems is software product lines that aim to decrease both development costs and development time. Before the concept of product line, analysis, design and development processes of software systems were handled just for a single product without the idea of reusability. Then, reusability was started to be implemented by copying and pasting code into different software products. However, it was realized that the method was insufficient for reusability of pieces of a software system and then, reusability of various parts of software systems such as assets, models, source code etc. becomes very important due to its stability, advantages on cost and time.

As the necessities, expectations and programming languages for developing software applications increase and change, various software development techniques and methodologies are proposed and implemented by various authors. One of them is feature oriented software development. This paradigm tries to generate different software systems that have both common and different features from that come from a set of predefined features. For such systems, feature oriented software development provides developing a family of products in a domain. It can be used for application engineering by selecting features and generating customized programs. The first step of the development technique is domain analysis. In domain analysis phase, the requirements and design choices are specified and then, the commonalities and variabilities of the domain are specified as features of this domain. A feature model

is designed with respect to the commonalities and variabilities specified in domain analysis. After analysis, domain is designed as sets of models or diagrams by using a modeling notation such as UML. In this phase, determining and designing the variability points of the domain is the most important issue for software development. After that, domain implementation is handled with various methods using aspects, feature modules and object-oriented programming. In the final step, the generation of a product or an application is accomplished according to the feature selections.

Object oriented programming concepts are used to build reusable and adaptable software, but most methods implemented do not indicate the variability of a domain and they are not domain oriented. Moreover, some methods concentrate on just software reusability by identifying and modeling reusable objects and most of object oriented methods have several problems in identification and definition of reusable objects and variability points. Structured analysis methods apply some guidelines for identification but it cannot capture the variability in related applications because the models are designed for generating a single application. For this reason, some authors try to use feature modeling in the development of object-oriented applications. Furthermore, a feature-oriented reuse method analyzes and models commonalities and variabilities in the applications of a specific domain for implementation technique features. It recommends a lot of guidelines in order to design an object model from a feature model. However, it assumes that an object model is derived perfectly, but these guidelines cannot cover all cases in object-oriented engineering. It does not also provide an automatic and stable variability management structure for object-oriented applications in a practical way.

The various implemented software product line techniques in literature are researched in the beginning of our study. During this research, we realized that the most important part of product lines is modeling the problem space. For mapping the problem space to the solution space, there are various modeling languages. Then, we concentrate on the issue of where, why and how feature modeling languages are used. In this research, feature-oriented software development (FOSD) paradigm is studied in detail. This development paradigm aims to provide construction and customization by using methods, design techniques, tools etc. All of these mainly aim to increase code reusability and manage the variability of large-scale software systems. In order

to achieve these goals, feature models try to map between features and all life cycles of software development. Key elements of these approaches are features that define the structure of a given program in order to satisfy customer requirements. That is, it can be supposed that features build a bridge between customer requirements and source code of a software system. In this manner, some authors use feature oriented programming in order to build this bridge. After detailed research, we focus on using the theory of feature modeling for object oriented engineering applications. We have a different point of view about this topic. Thus, we aim to implement a method providing reflecting the changes of configuration in cases that same application is used in same domain and so the application is not needed to be redesigned for each new configuration for the same domain. Our main motivation is to reflect the variability in a software company. And then, we decided to try to help generating an application that can work for all product combinations in a domain by using a feature model and so that the software system can be developed by using feature models. That is the reason why, customers can see the requirements specified by them and software developers can see the structure of the software system thanks to the same platform called feature model.

The main contribution of our approach is providing source code templates and data structure patterns to ease the code development process in a practical way and also reflecting automatically the variabilities in a domain to the object-oriented application obtained by using feature modeling. This aims to increase reuse of the software assets such as design and code while decreasing the need for redevelopment cycles with changes in requirements and decisions. In order to do that, our approach proposes a set of steps that are needed to be followed in a short time. In the first step, the variability points in a specific domain is determined as a feature model with respect to the demands of customers and specified design choices. And then, semi-automatic, time-saving and flexible domain-specific source code templates for developers to implement the application are generated automatically by our code generator. After the development of the application is completed, it is provided to work for all feature combinations by reflecting automatically the variabilities in this domain into the application through feature model. Thus, neither code replacement nor redesign of software is required for software developers and companies in order to apply various

configurations of a domain to an application and to use this application in the same field and so, the variability for the software system in a company can be reflected directly and automatically.

The content of this thesis is organized in the following chapters. In Chapter 2, definitions of various terms and technical information that are important to understand our work are explained. After this chapter, variability management and code generation methods, works and tools that can be related to our work in the literature are explained in Chapter 3. Then, our approach to variability management and source code generation in object oriented programming applications is explained with all details in Chapter 4. In the final chapter, namely, Chapter 5 a brief conclusion of our approach is described and what can be done in the future work is explained.

CHAPTER 2

BACKGROUND

Feature-oriented software development (FOSD) [4] is a paradigm that provides construction, customization and synthesis of software systems with various methods, languages, design techniques and tools for building variable, customizable, and extensible software. A feature is the most important piece of FOSD, since the aim of this paradigm is mapping between the representations of features and all phases of software life cycle, such as a domain engineer can trace features specified during the analysis phase with design and implementation.

There are many definitions about what a feature is and it is expressed in the next paragraph. In short, FOSD is a development paradigm that can be used to develop Software Product Line (SPL) and to do domain engineering. Software Product Line 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. [25]

2.1 Feature

Feature has not a unique definition in the literature; on the contrary, various viewpoints of what a feature have been appeared with respect to researches. That is, there are several technical and abstract definitions of a feature. For example:

1. The first definition is more technical than below definitions, "a structure that extends and modifies the structure of a given program in order to satisfy a stake-

holders requirement, to implement and encapsulate a design decision, and to offer a configuration option" [6], which has been first motivation and starting point of our study.

2. Zave [34] loads a technical meaning to feature, as well, "an optional or incremental unit of functionality"

3. The next definition is an abstract one that is "a triplet, f = (R,W,S), where R represents the requirements the feature satisfies, W the assumptions the feature takes about its environment and S its specification" [14].

4. This definition is more abstract than the previous one, "a logical unit of behavior specified by a set of functional and non-functional requirements" [13].

5. A completely abstract definition is "a distinctively identifiable functional abstraction that must be implemented, tested, delivered, and maintained" [21].

The technical definitions shows that features can be implemented in order to satisfy a software system's requirements that may represent design decision and provide a potential configuration option, but in the abstract definitions, it can be seen that features are used as abstract parts for specifying and separating software systems.

2.2 Feature-oriented Software Development (FOSD)

FOSD can provide domain engineering with the ability of planning and implementing SPLs. In addition to this, FOSD can be used for application engineering in order to select features and generate customized programs[31]. Feature discussed in the previous section is used for commonalities and variabilities during all life cycle of this process, so it is the main element of FOSD. To simplify, with respect to selection of features from a feature model, used FOSD technique will provide configuration options and the generation of software systems so that the software can be structured along these variabilities and commonalities. The four main phases composing the FOSD life cycle are [4]:

1. *Domain analysis* determines which features are included in the software system

or software product line. Feature models are very efficient in order to handle commonalities and variabilities, in other words feature models assist variability management. Features are main concept of all feature modeling approaches.

2. *Domain design and specification* phase defines the architecture of a software product line, whether in FOSD, it refers that a formal specification, an informal specification or a modeling language to specify the fundamental structural and behavioral properties of the included features. In fact, the main focus is feature modeling and feature implementation. The features are implemented when the features and their relationships are set in a feature model.

3. *Domain implementation* aims to set up a one-to-one mapping between features specified in domain analysis phase. There are various languages that try to provide features with better abstraction and modularization techniques. For instance, the Jak programming language extends Java and provides functionality so that a specific code mapped by a feature is stored in a specified directory[7]. Different from using abstraction and modularization, another idea is like C preprocessor to annotate code with features [22].

4. *Product configuration and generation* has a key role in FOSD. Feature selection of a user generates a working software system automatically. The first step of code generation is to provide tools in order to show the available features, their constraints and relationships clearly. For example, invalid feature selection is needed to be rejected during the feature selection for software system. Another step is to obtain a valid selection with respect to constraints of feature model, which will be discussed in future sections. The software needed to obtain is generated as soon as user have a valid feature selection.

The all four part of this theory have dependencies to each other like a chain. Although the process model of FOSD looks very simple for real-world systems, it can be used as a base in order to build methods, techniques for software product line engineering. Domain engineering and application engineering are the two main processes of SPLE. Actually, we can say that the first three phases of FOSD process are the concern of *domain engineering* and so the definition of this term is comprehensively discussed

in the next section. Moreover, product configuration and generation phase of FOSD resemble application engineering part of SPLE.

2.3 Domain Engineering

Domain engineering aims to develop domain artifacts that are used for developing applications for a given domain. A widely used definition can be given as "The process of software product line engineering in which the commonality and the variability of the product line are defined and realized." [27]. Due to the fact that, the most developed software systems are variants of other systems within the same field and not completely new systems, businesses can increase profits and reduce time-to-market by using the concepts and implementations from previously implemented software systems and applying them to the target system with the use of domain engineering [17].

Domain engineering focus on three main phases called *domain analysis*, *domain design* and *domain implementation* in parallel with application engineering. The outputs of each phase of domain engineering feed into both subsequent phases of domain engineering as well as corresponding phases in application engineering, which can be seen in the Figure 2.1 [2].

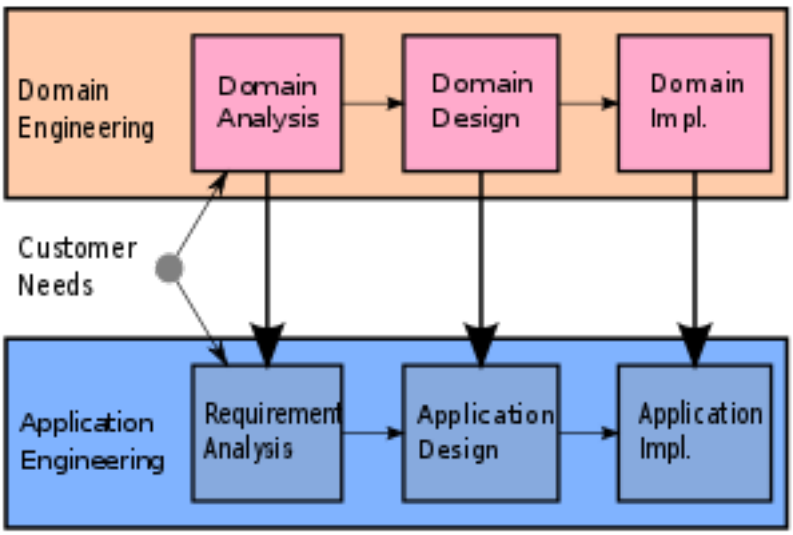

Figure 2.1: Domain Engineering and Application Engineering (adapted from [2])

In general, domain analysis is used to define domain by collection information about the domain and produce a domain model, but with Feature-Oriented Domain Analysis (FODA) method, domain analysis begin to aim identification of common points and varying points in the domain by using feature models.

2.4 Feature Modeling

We discussed about the definition of feature in the previous sections. In domain analysis, a feature model is used to describe both the features and the dependencies, relationships and constraints among the features of a specific domain. Moreover, feature models can be an input in order to produce requirements, documents, pieces of code etc. for SPL process. In 1990, firstly, Kang introduce and use the concept of a feature for the description of the commonalities and variabilities of software systems in the method of Feature-Oriented Domain Analysis (FODA) [20]. The main aim of this method is to endorse software reuse at the functional and architectural levels. In addition, a feature model can decide whether a combination of features in a domain is valid or not. This method is the basis of feature modeling among domain analysis. The software product line community has adopted many extensions and enhancements of feature modeling by means of FODA method as requirements increase.

In software product line implementation techniques, such as feature-oriented programming [28], delta-oriented programming [29], aspect-oriented programming [23] and preprocessors technique [22], feature models are needed and then they are widely used for commonality and variability management in software product lines. Various feature modeling notations specified in literature with respect to these different SPL techniques. We divide these notations into two categories, which are discussed in the following subsections.

2.4.1 Basic Feature Models

These model have a tree notation using each feature as a node in the tree. The main relation in basic models is between a parent feature and a child feature. Then, we can

categorize the relationships between a feature and their sub-features as four groups:

1. *Optional* represents that child feature is optional.
2. *Mandatory* represents that child feature is required. This means that a mandatory feature must be in all feature combinations.
3. *XOR (Alternative)* represents that just one of the child features can be selected.
4. *OR* represents that at least one of child features is must be selected. This means that one sub-feature or more can be selected alternatively.

In addition to the four relationships, defining cross-tree constraints between all features in a feature model is possible. Two well-known constraints are listed below,

1. X *implies* Y represents that when a random feature X from the feature model is selected, the selection of feature Y is mandatory.
2. X *excludes* Y represents that when a random feature X from the feature model is selected, both features can not be a part of the same product.

The basic feature model notations can be seen in Figure 2.2 adapted from [11]. The model denotes the features that generate the software of mobile phone. *Calls* feature is mandatory for all product domain of mobile phone according to the model and the software for mobile phone provides optionally *GPS* feature and *Media* group including *Camera* and *MP3* features. Furthermore, the one of the three screen mode choices can be selected for the screen aspect of the mobile phone. The two cross-tree constraints can also be seen in the figure.

2.4.2 Extended Feature Models

Different from basic feature models, there are extended feature models like cardinality-based feature models and alike. For example, basic feature model with the [n, m], where n is the lower bound and m is the upper bound, form of relationship like UML.

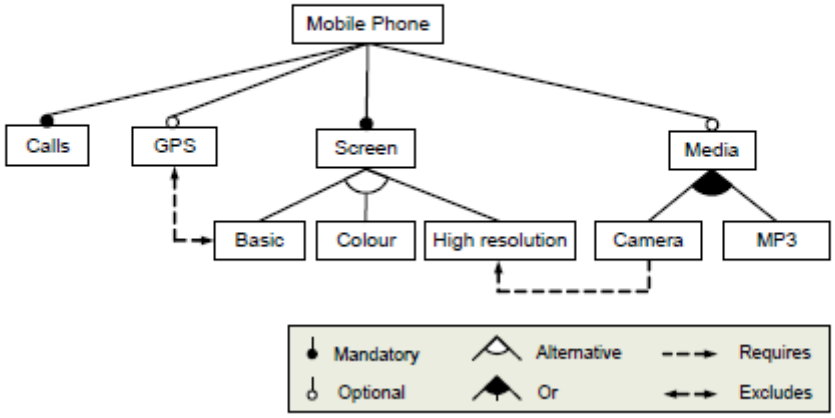

Figure 2.2: A basic feature model sample (adapted from [11])

Through the two number, when a parent feature is selected, the number of child features, which can be a part of product, can be limited [15]. Moreover, some feature models that try to handle functional features using attributes included a name, a domain and a value, proposed by Benavides [12].

To give an example for extended feature models, FeatureML [18] gives features two different main meanings with respect to implementation of features in the source code. The first one is *Data Feature* that has language elements for primitive features, properties for parent-child tree structures and inheritance logic of object oriented design. That is, this feature feature modeling language assumes that entities and properties are represented as data data features in the feature model. The second feature type is Behavior Feature that has dependencies to other features in both types in feature model and text information about features. Different from basic feature models, an acyclic graph is used in order to represent features in FeatureML.

2.5 Feature Diagram

Feature diagrams are widely used for representing a tree-sructured feature model visually. That is, a feature diagram can be used as a graphical hierarchy of features

and to cover composition and generalization of a feature model. The relationships between a parent feature and their child features mentioned in feature modeling section can be represented in graphical notations like in Figure 2.3 that is adapted from [8]. There are various feature diagrams used in software product line systems as tree notation, whereas some authors choose various graph notations for some extended feature models like Guozheng [18].

Figure 2.3: Feature Relationships in Feature Diagram (adapted from [8])

We realized that when a feature diagram notation is good at visuality, it can be more user friendly for domain experts in order to generate a feature model. An open-source and feature-oriented software development framework FeatureIDE [32] can provide a simple and comprehensible interface for both domain experts and their customers like in Figure 2.4. Feature modeling notations, cross-tree constraints and abstract features can be noticed in this feature diagram. In addition to this, some functions such as adding a new feature or a new constraint to the model and changing characteristic of a feature is very simple for users by this tool.

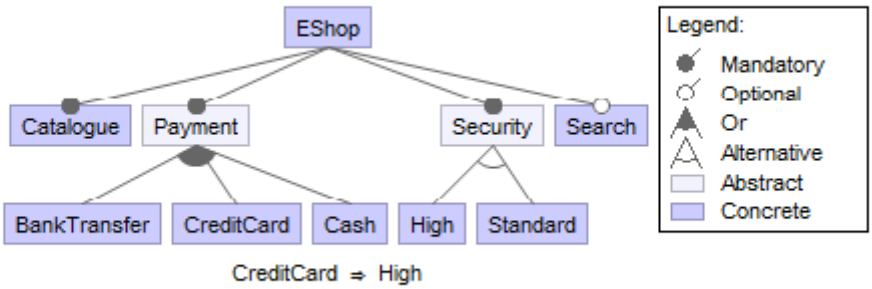

Figure 2.4: Feature Diagram Sample in FeatureIDE Tool

2.6 Configuration

When feature selection process is ended, the product is generated in terms of features in the feature model. This subset of all features in the feature model is called*Configuration*. A configuration is valid whenever it can achieve the cross-tree constraints and the semantics of rules of the feature model. If it can not achieve all, the configuration will be invalid [32]. Figure 2.5 shows a valid configuration with the number of possible combinations for the model in Figure 2.4.

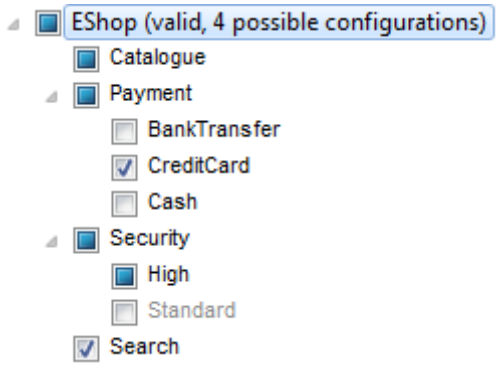

Figure 2.5: A configuration sample in FeatureIDE tool

2.7 Software Commonality and Variability

The concept of *Commonality* represents that the functional or non-functional aspects are common for all products in SPL platform. Whereas,the definition of *variability* is the characteristics, which needed to be explicitly modeled and implemented so that selective inclusion in products of a domain is possible, can change in some products. In this sense, *Software Variability* is conduciveness of a software system or artifact to be efficiently extended, changed, customized or configured for use in a particular context [30]. For example, executable files containing alternative modules such as for integer, double and floating point arithmetic, source code including alternative code parts like preprocessor directives, classes implementing the same interface but having different implementations, etc.

2.8 Variability Management

We mentioned that SPL is a set of software-intensive systems sharing a common and managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [25]. The common set of features in a feature model symbolizes the commonality of a software system, while the managed set of features in the model symbolizes the variability of the software system. Variability management is one of the fundamental principles of software product line management. Individual systems are intended as variations of a common theme. This variability is needed to be straightforward, explicit and systematically managed. Moreover, both commonality and variability of SPL must take into account some factors such as technical restrictions and choices, business and technology strategy, market and customer needs, ... etc, so the future challenges can be reduced in all SPL life cycle.

CHAPTER 3

RELATED WORK

In this chapter, various variability management and code generation methods will be separated into groups in terms of the techniques of variability modeling and code generation they used, and the details of these techniques will be explained with examples. Moreover, both the advantages and the disadvantages of example methods and tools are explained.

The code generator must have a connection with a feature model, since design choices selected from the defined feature model must be used during the code generation process. Establishing this connection is one of the most challenging parts for this process and our work. In this manner, we need to handle and specify feature modeling and code generation process together. Therefore, some researchers focus on details of feature modeling and how to manage variability. In the literature, various level and structure of feature modeling techniques are promoted by using graphical or natural languages. Various feature modeling techniques are divided into groups and explained below. The one of these groups is the models based on Unified modeling language (UML) [19]. The variability modeling methods add variability notations such as tagged values, stereotypes, constraints, etc. to UML notation due to the variability management. Ziadi et al. [36] proposes a UML profile containing stereotypes, tagged values, and structural constraints in addition to product line models with variabilities. Since UML is well known by software designers, the variability models extending UML can be learned more easily than the other variability modeling languages by them. The weak side of UML extension models is that variability information of features is in the first class entities in a UML diagram instead of a feature model, because

this situation complicates analyzing and tracking the variability of software system.

Another feature modeling style is Orthogonal Variability Modeling (OVM) that is used by Gamze [10] in order to manage variability by tagging the information of variability in the domain with configuration management part and adding the version information of software system. This technique can be useful for long-running and big-scale software systems to create various family models, relationships between them and the information of version. On the contrary, our approach can be useful for middle-scale or small-scale software systems that are needed to be developed in short time periods. Moreover, this method may be very suitable for software companies that want to develop a single application for a specified domain rather than developing various applications for this domain. Our approach can provide to reflect the variability in a domain automatically and directly to a single application for such companies in a short time period.

Software product lines set with UML based feature modeling extensions concentrate on variability in terms of high level designs of product families. However, they concentrate on variability in terms of neither customization nor composition and generalization of software applications. Different from this style of approach, we wanted to use more basic variability modeling with trying to constitute mapping patterns between these models and code generation in our work, since a product line must have a common and customizable variability modeling notation so that all phases of feature oriented software development can correspondingly progress.

The second group is the models based on Extensible Markup Language (XML). For example, XFeature [3, 26] is a pure XML based product family modeling tool. This GUI based tool provides users to describe a model representing the set of applications of product family that a set of applications have reusable software assets. This framework using various XML technologies provides users both feature modeling and variability modeling. The style of graphical elements resembles the style of feature diagrams in Feature-Oriented Domain Analysis (FODA). Additionally, this tool support cardinality for feature modeling. Another XML based tool becoming popular is XVCL [35] using XVCL language. In this tool, commonality and variability in a product family are symbolized as x-frames that is command templates such as adapt,

insert, delete, replace, etc. XVCL frame manage the variability by using SPC elements including the information of variable values and design choices and providing software components like requirements and pieces of source code.

There is another modeling approach based on Component Oriented Software Engineering (COSE). The paradigm of COSE has many differences from object-oriented programming. In object oriented design, the main idea is to model the real world objects and create nouns and verbs so that developers can code them. On the contrary, software systems are composed from software components in COSE. According to this paradigm, development by integration of software components is suggested instead of writing code pieces. COSEML using this approach is a graphical component oriented software engineering modeling language [16]. This modeling language includes components representing structural piece of software rather than basic objects in the software.

We mentioned that managing the variability ensures substantially the connection between feature model and the code generation process. There are several code generation techniques using the mentioned variability modeling languages and variability management methods in order to implement software product lines in various fields. We divide these techniques into groups with respect to code generation and variability management styles.

Firstly, assembler based code generators select the each matching source code part from the specified source code pool with respect to each mapping between the software requirements and specifications and prepared source code pieces. The translation queries for each mapping between source code files and requirements search and generate the final source code files. For example, Zhang and Jarzabek develop product line assets as a set of x-frames managing commonality and variability in a domain by using XML-based Variant Configuration Language (XVCL), a variability mechanism based on frame technology [35]. According to this approach, both all possible design choices and corresponding source code components are prepared initially. After that, code components are selected automatically for each feature combination from a feature model. That is, a selection of a feature stands for that selection of a piece of source code in this approach. However, XVCL is not scalable enough and

preparing source code for all possibilities costs too much as the number of features increase. Another assembler based work proposed by Batory is AHEAD tool suite using feature oriented programming (FOP) method [9]. The basic idea of FOP is incrementally building simple programs by adding features, so each added feature to the feature model has a simple program fragments described as Jak feature files using jak language that extends Java programming language and support code generation with java, but not interested in feature variability and dependencies. An example flow of AHEAD process in Figure 3.1 can represent how the tool generates source code. The number of N times jak files are initially developed by software engineers. The N jak files form into a single jak file by using the composer of *jampack* or *mixin*. Then, this generated single jak file is translated into java source code files by using a translator named *jak2java*. In order to manage feature variability, Batory proposes a method that features in a feature model and aspect-oriented programming with FOP can be merged in terms of aspects [9]. This method tries to generate source code by integrating the aspect of data modeling into feature model with various code generation patterns. Similar with AHEAD, FeatureC++ extends syntax C++ programming language with different and new keywords. With command-line tools, this tool composes feature oriented C++ files in order to generate source code files [5].

Secondly, template based generators uses tag mechanism in which tags placed into source code are related with system requirements. Then, generator replace the specified tags in the source code with the source code, which is previously related the requirements. Rhizome [18] is both template and assembler based code generation platform. This platform has a template language called MarkerML and a feature modeling language called FeatureML. In order to see how this platform manage variability and generate code, we try to cover the workflow of Rhizome in Figure 3.2. As we can see in figure, feature model designers constitute a feature model after they select design choices from feature design wizard of product line space. After that, platform developers try to specify the template of source code and the software variants of software product line that they work on. With the specified variants, these developers try to cover code structure commonality and variability in order to write templates using tags defined in MarkerML for template repository and in this way; they implement various code generation patterns as text structures and model them as code generation

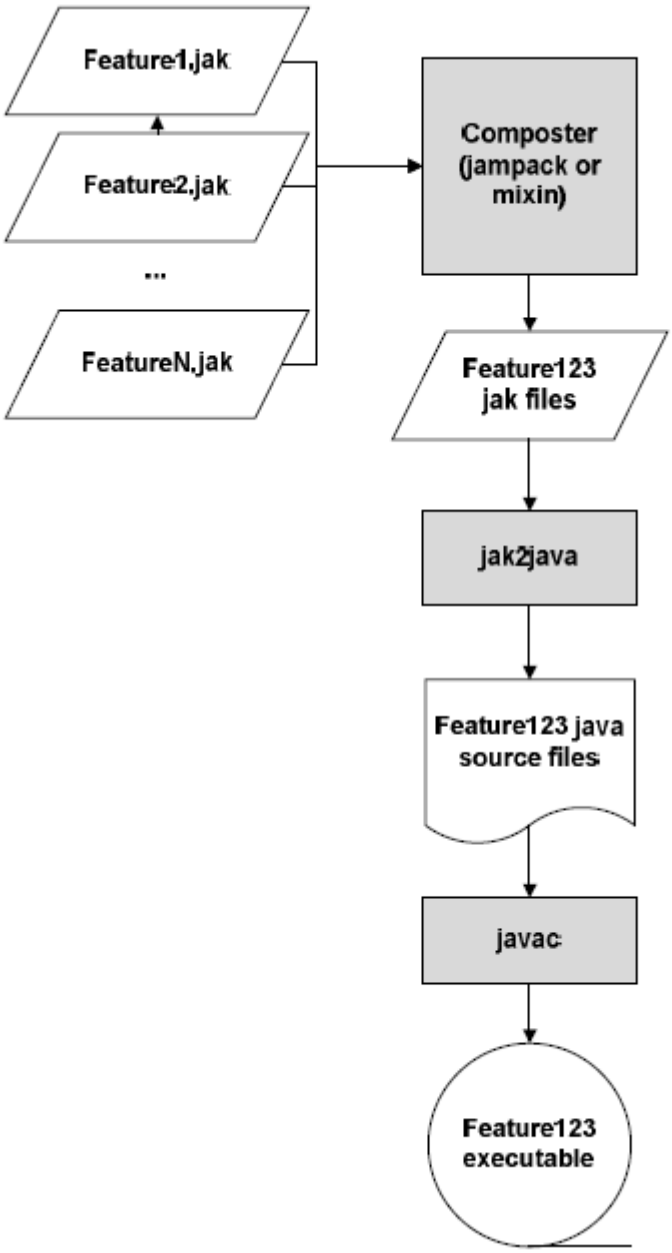

Figure 3.1: An Example Workflow of AHEAD Toolkit (adapted from [9])

unit (CGU) types. At final step, the code generator uses feature design selections as parameters for creating source code blocks and then, uses the written template files in order to handle the variability by translating CGUs into generated source code blocks during code generation process. Rhizome tries to generate completely working applications by generating all source code for all configuration combinations. In order to do that, the platform developers must study commonality and variability of the system and then must prepare template files for all combinations and it can be very costly as

the variability of the product line increases. In our work, we use advantages of object oriented design concepts to reduce the cost of such cases by our code generation and variability management patterns.

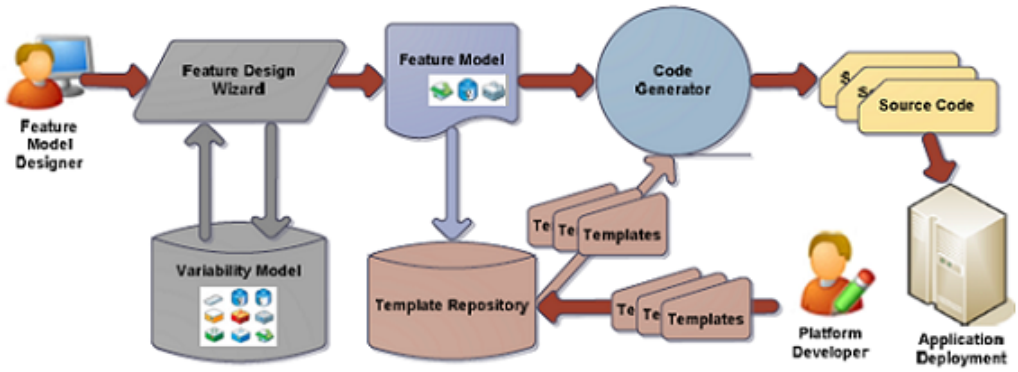

Figure 3.2: The Workflow of Rhizome Platform (adapted from[18])

Different from the previous approaches, Lee et al. extended the feature-oriented reuse method called FORM that analyze and model commonalities and differences in the applications of a given domain with respect to implementation technique features used for deriving objects related with the features and develop reusable and adaptable domain architectures [24]. They define his approach that a feature-based object oriented engineering method. They firstly analyze applications in a domain in terms of features and then, determine candidate objects based on feature analysis. After this operations, FORM analyzes the relationships between features in the feature model in order to construct the object model as it can be seen in Figure 3.3. During the identification of development objects from feature model, feature and object categories are specified and set up relationships between them by software designers with respect to the specified guidelines of FORM. The generated object model supports the three main object relationships: aggregation, generalization and associations. That is, this method does not provide a code generation mechanism, but it helps the generation of object model with respect to a configuration of a feature model by recommending pre-prepared guidelines.

Recently, a new SPL implementation technique is developed called *Preprocessors* [22]. This technique provides a conditional compilation in order to manage variability

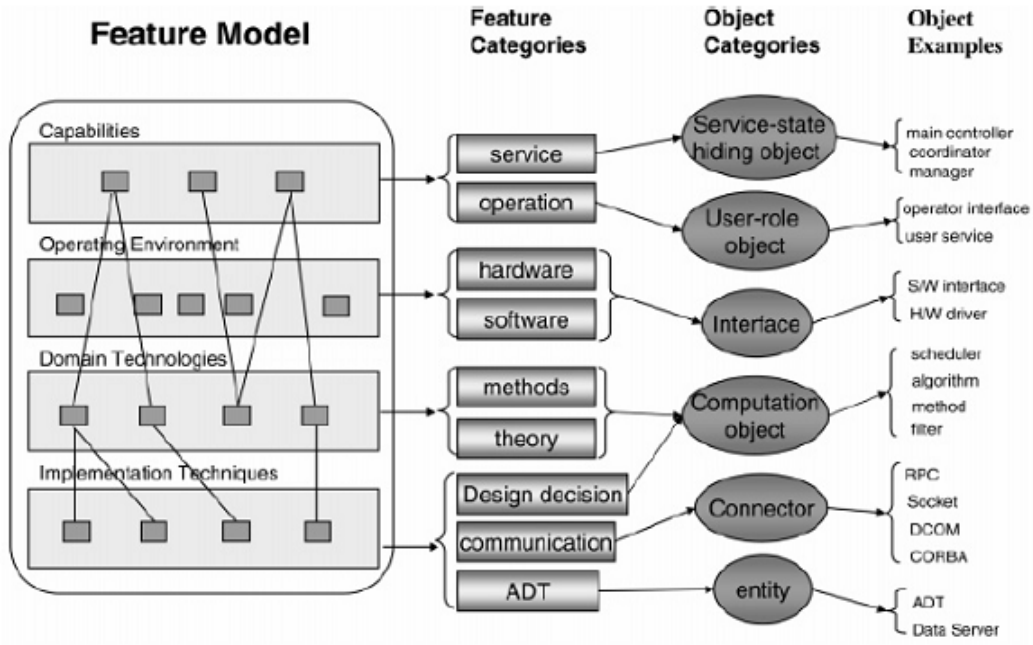

Figure 3.3: The Workflow of FORM Extension Method (adapted from [24])

in compile-time. FeatureIDE provides Antenna and Munge that are preprocessor languages for Java programming language infrastructure in order to implement SPL with FOSD theory [32].

CHAPTER 4

MANAGING APPLICATION VARIABILITY

General life cycle of object oriented programming has three main steps. The first step is to determine the requirements and the possible design choices of a software system. Then, the selected design choices are transformed into various models, such as UML, feature modeling language and etc. including interfaces, components and relationships between the software components in the second step. In the last step, all source code of the software system is implemented and developed by software engineers with respect to the generated model of the system. In this work, our first aim has been to generate source code templates by using feature modeling for applications implemented by object oriented programming, so that developers do not need to create the initial classes and interfaces of the system and they consume less time to implement source code and make connections between the code artifacts in the generated code. Our second aim is as a final product to obtain an application that is to be able to work automatically for all combinations of design choices of the software system without any change on the system. This idea has grown with our research that how software product line techniques are implemented. As explained in Related Work Section, we realized that various levels of product lines uses very different techniques for feature modeling and product generation. There are different tools providing complete source code generation or managing variability for product families. However, we could not see a work about generating template source code and developing just one product family that can work for all combinations of features in a feature model in object-oriented engineering domain. In the light of this literature research, we decided to implement our methodology to give a different viewpoint for implementing object oriented applications.

The workflow of our methodology can be seen in Figure 4.1. After the feature model is designed according to the design decisions in the domain by using a feature modeling tool, class templates and configurator management class is generated by using the information of this feature model as the first step of workflow. The details of code generation part will be given in the following sections. In the next step of our workflow, developers continue development of the object oriented application by using the class templates. After the development process is finalized, a configuration is specified by selecting the desired features in the feature model. This configuration file must conform the rules and cross tree constraints of feature model. The application takes this file containing the names of selected features as an input and the information in this file is sent to configurator management class. After that, this class make required bindings in the code to provide automatically dependency injection of the application in run-time process. When the configuration file is changed by user to select different features, the application can run with respect to the new selected features. Thus, Figure 4.1 can demonstrate that the variabilities in the domain can be reflected automatically and directly to this object-oriented application with the feature model.

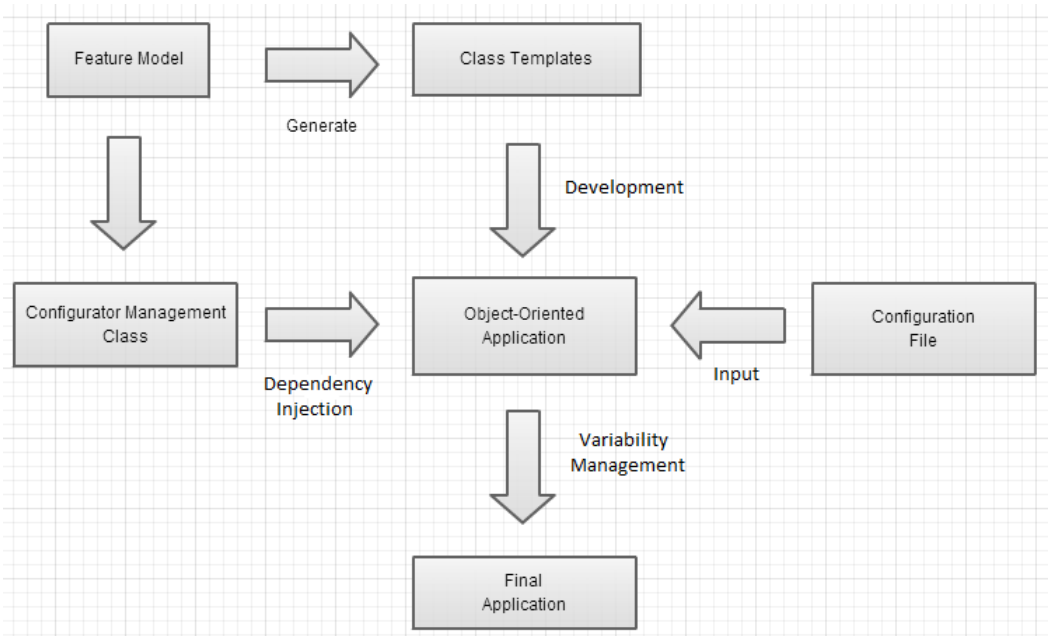

Figure 4.1: The Workflow of Our Approach

In this chapter, the details of our methodology is explained in five sections. The organization of this chapter is as follows: first of all, feature modeling language used

in our approach will be detailed, and then in the next section, basic assumptions for phases of this study are listed in order. In the third section, how we parse the feature model in order to gather information about features and their relations with the other features are clarified. Then, how we generate source code templates by using Object Oriented Design techniques are explained in the next section. In the last section, the method managing the software variability of E-Shop application family with inversion of control is described.

4.1 Feature Modeling Language: FeatureIDE

We mentioned the definition and usage of feature model in detail in previous chapters. We emphasized that a future model can be used to represent the valid combinations of features in a domain. In our work, a feature modeling tool is needed to structure the domain and we decided to choose FeatureIDE due to its simple and user friendly interface for feature modeling as a feature diagram. In addition, this tool provides to generate a valid feature configuration with a configuration interface for both domain experts and software engineers. FeatureIDE allows various connections between a feature and its subfeatures as a simple feature model sample models a result of online shopping domain analysis in Figure 4.2. We can easily create a feature model by using just mouse of computer with well-organized GUI of this tool. Furthermore, as it can be seen in this figure, we can see graphically all details of E-Shop feature diagram. It can show the connections between features used in feature model such as *mandatory* connection between 'EShop' and 'Catalogue' features and *optional* connection between "EShop" and "Search" features, *And* group connection between "EShop" and its all subfeatures, *Or* group connection between "Payment" and its subfeature group and *Alternative* group connection between "Security" and its subfeature group as a legend. The connections of *Mandatory* and *Optional* can be used for the subfeatures of *and* group connection, but the two connection type cannot be used for subfeatures of *Or* and *Alternative* group connections. For example, you need to specify the feature of "BankTransfer" or "High" as neither *Mandatory* and *Optional*. In addition to these options, cross tree constraints can be defined smoothly and added multiple. For example, *Requires* constraint between "CreditCard" and "High" features and *Excludes*

constraint between "Standard" and "Search" features.

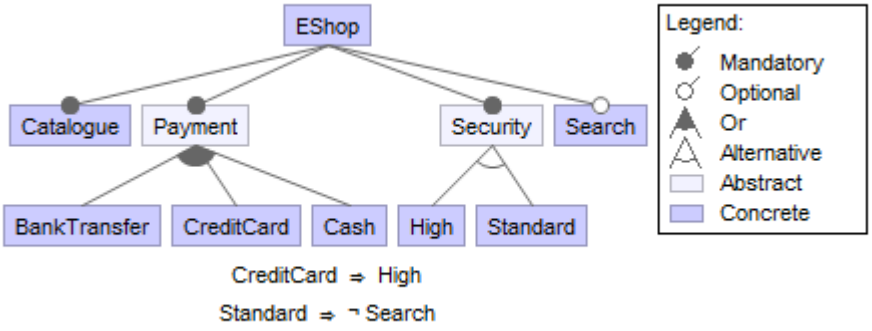

Figure 4.2: A Sample Feature Diagram in FeatureIDE

FeatureIDE serves two types of feature called *Abstract* feature and *Concrete* feature in addition to above connections and constraints. Thomas et al. stated that if a feature will used to structure a feature model, but this feature does not affect the implementation process [33]. They emphasize that abstract features has an important effect during domain analysis to be able to symbolize domain decisions that do not have any action about the generation of a software program variability. We use abstract features by in order to generate types and save the structure of the feature model by taking into account the thoughts of them. The usage of abstract feature in our approach will be explained in detail in the next chapter. If a feature is not abstract, its type must be *Concrete*. A *Concrete* feature has at least one mapping with implementation components during both code generation and variability management level.

The final state of a feature model includes all constraints and relationships between features. We showed that FeatureIDE can represent visually them for users and all information of the constituted model is written with the tags of features in an XML file. When the menu item of *Source* in feature diagram menu of the tool is clicked, the XML notation of this file can be seen like in Figure 4.3. This recursive tree notation of XML file is very useful in order to process the feature model, because the relationships between a feature and its child features and the constraints defined between features can be handled and managed by basic XML Parsing methods. In our approach, we used *Java DOM Parser API* that builds a tree representation of the entire document with nodes in order to parse our XML file of feature model.

```xml
<featureModel chosenLayoutAlgorithm="1">
    <struct>
        <and mandatory="true" name="EShop">
            <feature mandatory="true" name="Catalogue"/>
            <or abstract="true" mandatory="true" name="Payment">
                <feature mandatory="true" name="BankTransfer"/>
                <feature mandatory="true" name="CreditCard"/>
                <feature mandatory="true" name="Cash"/>
            </or>
            <alt abstract="true" mandatory="true" name="Security">
                <feature mandatory="true" name="High"/>
                <feature mandatory="true" name="Standard"/>
            </alt>
            <feature name="Search"/>
        </and>
    </struct>
    <constraints>
        <rule>
            <imp>
                <var>CreditCard</var>
                <var>High</var>
            </imp>
        </rule>
        <rule>
            <imp>
                <var>Standard</var>
                <not>
                    <var>Search</var>
                </not>
            </imp>
        </rule>
    </constraints>
    <comments/>
    <featureOrder userDefined="false"/>
</featureModel>
```

Figure 4.3: The XML Notation of An Example Feature Diagram in FeatureIDE

After a feature model is constituted by a domain engineer, various product configurations can be created and hold as a combination of the features in the feature model by using the configurator of FeatureIDE. After the feature model is completed, the configurator will be ready for feature selection of a product. It can show also the information of the number of possible configurations as it can be seen in Figure 4.4. The left configuration file in Figure 4.4 shows "17" for this number. Furthermore, the four mandatory features called "EShop", "Catalogue", "Payment" and "Security" are shown already selected in this file. When "CreditCard" feature is selected, it can

be seen that "High" feature is selected automatically by the configurator due to the cross tree constraint of *Requires* between these two features. When "BankTransfer", "CreditCard" and "Search" features are selected, the configurator calculates again automatically the new number of possible combinations as "2" in the right configuration file in Figure 4.4. A configuration file includes the list of names of selected features in order except abstract features.

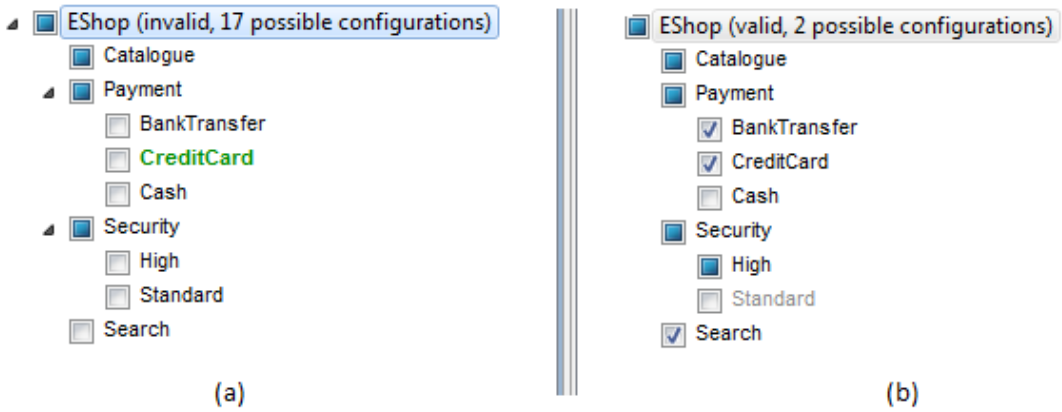

Figure 4.4: An Invalid (a) and A Valid (b) Feature Configuration in FeatureIDE

4.2 Basic Assumptions in Our Approach

Before the detailed information of our study is given, the basic assumptions made especially for domain analysis phase of our approach can be listed below. These assumptions may be important and useful for domain engineers that want to use our approach in object oriented applications.

- The feature model prepared by domain engineers during domain analysis is supposed to be well defined and formed. The requirements are also supposed to be specified very well in feature model.
- Each feature in a feature model has a unique name. This situation about names is very important, since data structures and mappings are set up with respect to the names of feature. A conflict with the names of features in a feature model damages all processes in our approach. Thanks to FeatureIDE conflicting about the names of features is prevented while the feature model is being developed.

In cases that this tool is not used for feature modeling, domain engineers should pay attention significantly to this point.

- Basic naming conventions of Java programming language are supposed to be used to determine the names of features. For example, invalid characters must not be used in feature names because they will be used in source code generation of Java classes. If an invalid name is set to a feature name, FeatureIDE gives warning, "The name need to be a valid Java identifier" and then it does not allow the invalid name.

- Since an abstract feature and a concrete feature have different impacts on data structures that will be used in code generation, abstract features are needed to be made explicit during feature modeling by domain engineers and developers. Thus, abstract features in the feature model are supposed to be made explicit for both feature diagram and the XML file of feature model.

- Concrete features in a feature model are mapped by main entity classes and abstract features are mapped by interface types in implementation level of our approach. We supposed that domain engineers and software developers are aware of how the mappings between feature model and source code are related and so they finalize the tree of feature model with respect to the mappings. Developers are supposed to develop an application family by considering the data structures explained in the next section.

4.3 Recursive Parsing of Feature Model

The first main step of our approach is parsing the XML file of feature model recursively after the feature model of a product family is fulfilled. We have implemented a simple algorithm based on depth-first traverse in order to create a tree structure of all features. This algorithm traverses the each feature of tree in depth-first order and it performs the following operations recursively at each node:

1. Perform pre-order operation
2. For each i (i: from 1 to n-1) do:

2.1. Visit i-th, if present

2.2. Create a feature node with its properties

2.3. Perform in-order operation

3. Visit the last (n-th) child, if present

4. Perform post-order operation

where n is the number of child nodes. This algorithm structures all features in a feature model to create a tree structure including entity classes with object-oriented design. Firstly, it puts the root feature into the beginning of the tree and then, the below subfeatures of the root one are visited recursively. For a compound feature that has child features, a parent feature is specified as a node with the information of whether a feature is abstract or concrete and the information of relationships between the parent feature and its subfeatures according to the connections between features such as *And*, *Or*, *Alternative*, etc ... Then, each child feature is specified as a node with the information of its parent feature in sequence. Moreover, the tree structure, which all nodes are features of a feature model and containing meaningful information for the following phases of our method, is used to generate source code templates including classes with specified attributes, interfaces and the extra smart classes that will manage variability of the final application. The details of the source code artifacts are explained in the next section.

4.4 Generating Source Code Templates with Data Structures

We realized that object oriented programming concepts are very useful for transformation of feature model to code artifacts and then these code artifacts can be used in order to be able to manage the variability of feature modeling with respect to the selected design choices in run-time. In our approach, we concentrate on customization, generalization and composition of software applications with the logic of object oriented design. In this manner, the tree structure of features generated with our XML parsing method is explained in the previous section. During this process, interfaces as types, source code template classes that include data structures changing with respect

to the connections between features and then, the configurator classes that will allow to manage variability of application with respect to concepts of OOP design are generated. We found various code generation procedures in Java programming language in the literature after some research. We choose *CodeModel API* [1] for the code generation process, since generating source code is a seldom required use-case which typically implies that library support is lacking and the *CodeModel API* provides excellent support for generating Java source code in a type-safe fashion. Moreover, all import statements are automatically added to Java files thanks to CodeModel. This API is also very compatible with our XML parsing method. We divide the generated source code templates into two groups. The main entity classes and interfaces generated in terms of the feature modeling connections are detailed in the first subsection and then, the configurator classes are clarified in the next subsection.

4.4.1 Entity Classes and Interfaces with The Content of Data Structures

In the previous sections, we concentrate on feature modeling side and parsing of feature models, where a domain engineer designs the model by specifying various design decisions about which specific design options is made for each feature. We stated that we used FeatureIDE tool for feature modeling. This tool can store all these design options in textual form. On the other hand, we needed a method for source code generation by using this textual information. We have a mapping connector including Java classes in order to create mappings and using our XML parsing method in order to figure out the connections of features in the feature model.

4.4.2 *Mandatory* Connection

The first connection between a feature and its child feature is *Mandatory* that means a mandatory feature always exists for its parent feature. A feature model containing just two features with this connection between "FeatureA" and "FeatureB" represents the concept of composition in OOP design as it can be seen in Figure 4.5.

Our code generator gives the meaning of composition of "FeatureB" in "FeatureA" and this connection in the feature model is mapped to a UML class diagram in Figure

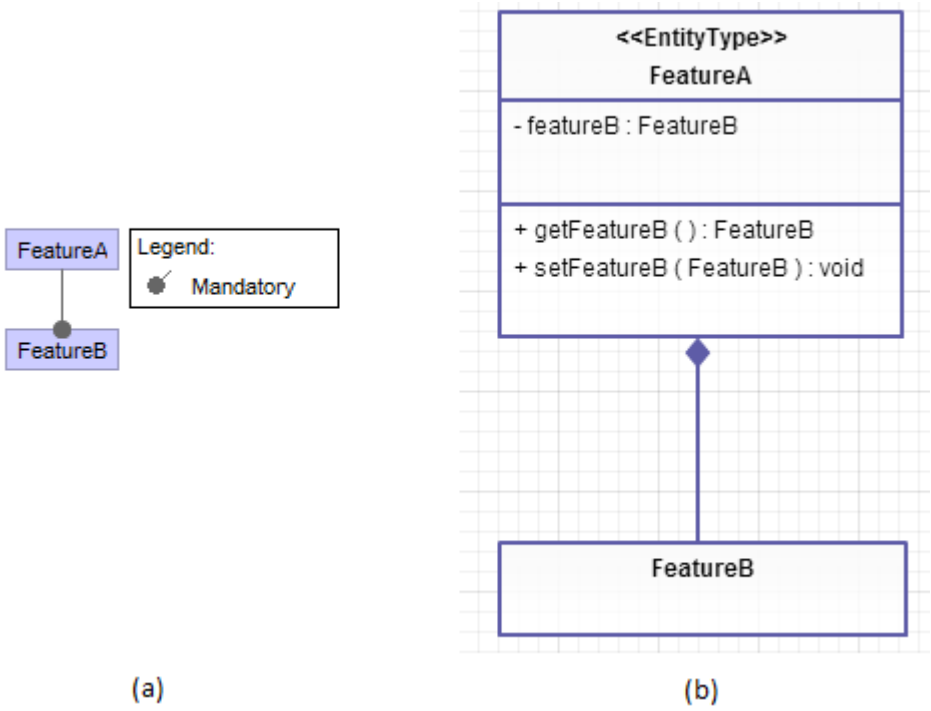

Figure 4.5: A Mandatory Feature (a) and The Corresponding UML Class Diagram (b)

```
public class FeatureA {
    private FeatureB featureB;

    public FeatureA() {
    }

    public FeatureB getFeatureB()
    {
        return this.featureB;
    }

    public void setFeatureB(
            FeatureB featureB)
    {
        this.featureB = featureB;
    }
}
```

```
public class FeatureB {


}
```

Figure 4.6: Generated Source Code for Mandatory Connection

4.5 and then it generated the entity Java classes namely, "FeatureA" and "FeatureB" with getter and setter methods in Figure 4.6.

4.4.3 *Optional* Connection

The second connection is *Optional* relation like between "FeatureA" and "FeatureB" in Figure 4.7. For this connection type, our code generator gives the meaning of composition of "FeatureB" in "FeatureA" similarly for *Mandatory*. The result UML class diagram of the feature model in Figure 4.7 is as same as the feature model containing just *Mandatory* connection in Figure 4.5. As it can be seen in the two figures, we consider *Optional* connection as identical of *Mandatory*, since when the optional feature is selected, it does not have any difference from a mandatory feature for their parent feature. That is why, the optional feature case is implemented by our generator and then,

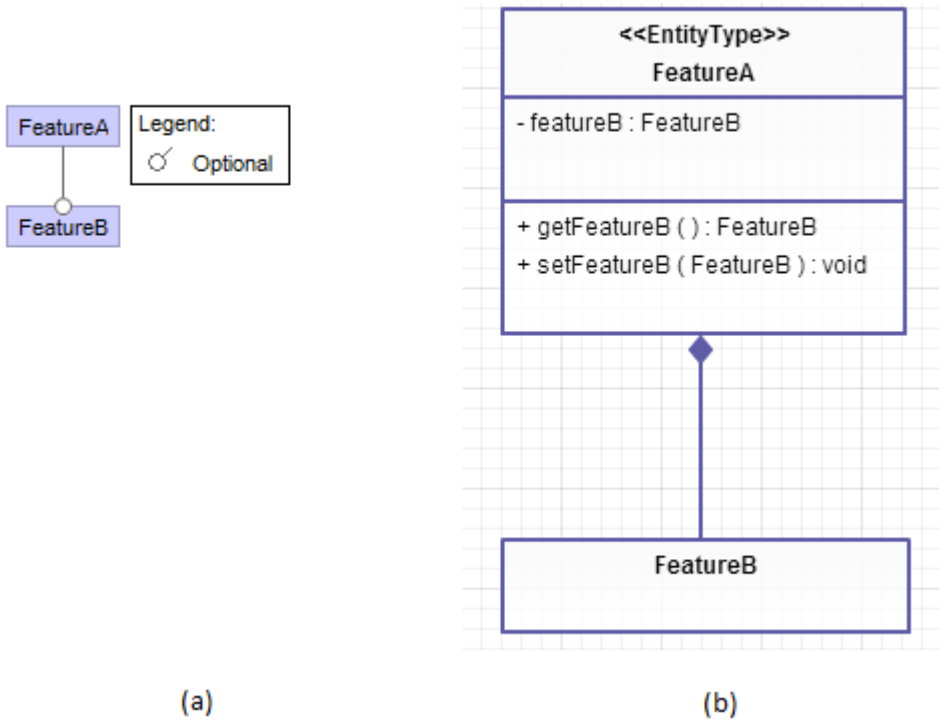

Figure 4.7: An Optional Feature (a) and The Corresponding UML Class Diagram (b)

our configurator classes checks whether the optional feature is selected or not in the product configuration in run-time to manage the variability point. If the optional feature is selected, our configurator will create an instance of "FeatureB" class and set the attribute, which type is "FeatureA", in the instance of "FeatureA" as the new created instance by using a statement, *"featureA.setFeatureB(new FeatureB());"*. Otherwise,

the configurator will set this attribute of the instance with "FeatureA" class type as null. On the contrary, a mandatory feature will always be set a new object, because it is in all product combinations.

4.4.4 *And* Connection

When a feature has child features, the type of the feature can be one of *And*, *Or* and *Alternative*.

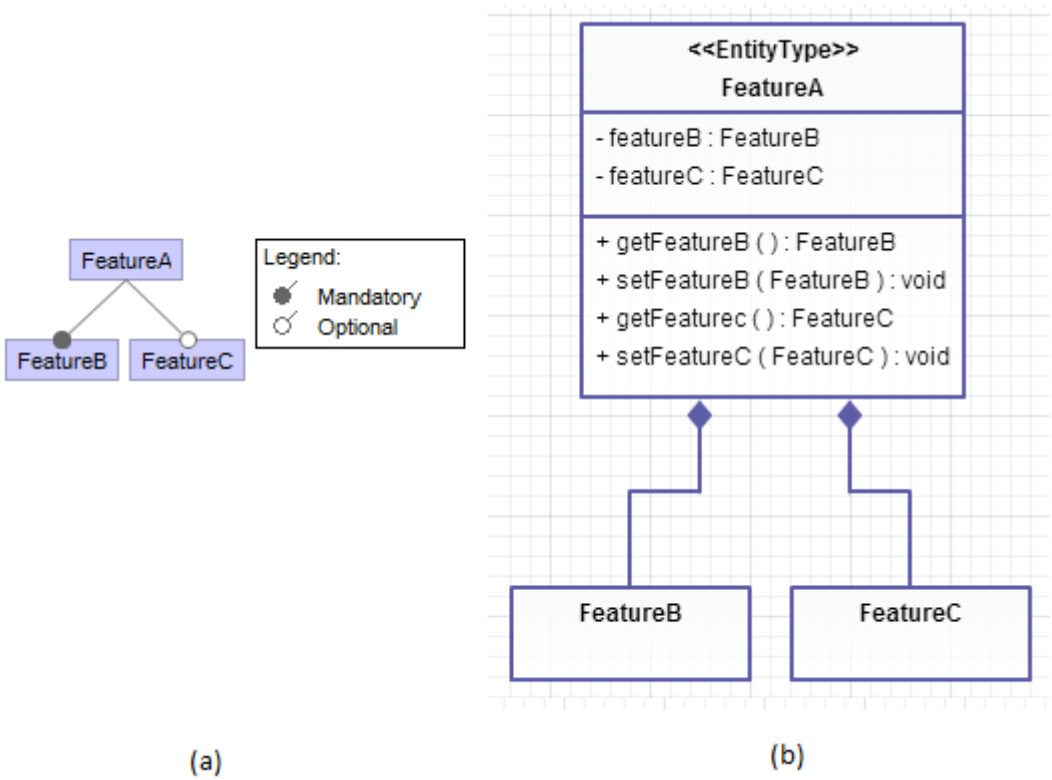

Figure 4.8: The *And* Feature (a) and The Corresponding UML Class Diagram (b)

The connection type between "FeatureA" and its subfeatures in the feature model in Figure 4.8 (a) is neither *Alternative* nor *Or* and so it is an *And* connection used for a relationship like in this feature model. This type of connection symbolizes composition logic in feature modeling. We also use it as a composition between "FeatureA" and its subfeatures that can be either a *Mandatory* or an *Optional* type. The class diagram in Figure 4.8 represents the generated template code for *And* connection. This diagram can be thought as a union of the class diagrams of *Mandatory* and *Optional* features explained in previous sections.

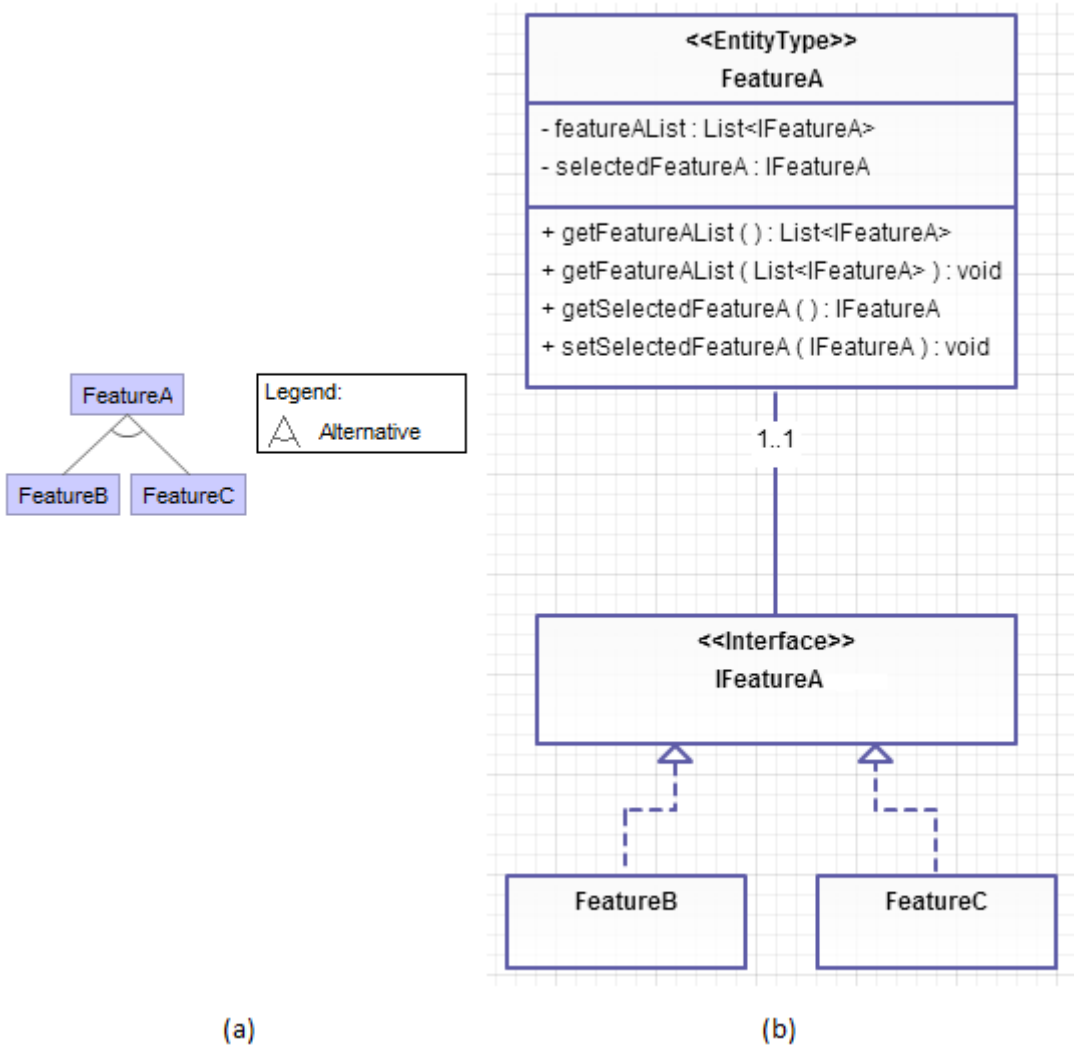

Figure 4.9: The *Alternative* Features (a) and The Corresponding UML Class Diagram (b)

4.4.5 *Alternative* Connection

The root feature, namely "FeatureA" has an *Alternative* connection with "FeatureB" and "FeatureC" in the feature model represented in Figure 4.9. For the connection type of *Alternative*, our code generator creates firstly an entity class of concrete "FeatureA" without fields and methods and then, creates an interface called "IFeatureA" by using the parent feature of this connection. After the generation of this interface, the entity classes of concrete subfeatures "FeatureB" and "FeatureC" are generated. Our generator makes the classes of subfeatures implement the interface generated

with the type of "IFeatureA" just before these classes are created.

```
public interface IFeatureA
{

}

public class FeatureB
    implements IFeatureA
{

}

public class FeatureC
    implements IFeatureA
{

}
```

```
public class FeatureA {

    private List<IFeatureA> featureAList;
    private IFeatureA selectedFeatureA;

    public FeatureA() {
        this.setFeatureAList(
                new ArrayList<IFeatureA>());
        this.getFeatureAList().add(new FeatureB()
        this.getFeatureAList().add(new FeatureC()
    }

    public List<IFeatureA> getFeatureAList() {
        return this.featureAList;
    }
    public void setFeatureAList(
            List<IFeatureA> featureAList) {
        this.featureAList = featureAList;
    }
    public IFeatureA getSelectedFeatureA() {
        return this.selectedFeatureA;
    }
    public void setSelectedFeatureA(
            IFeatureA selectedFeatureA) {
        this.selectedFeatureA = selectedFeatureA;
    }
}
```

Figure 4.10: Generated Source Code for *Alternative* Connection

The UML Class Diagram of generated source code in Figure 4.10 for this feature model in Figure 4.9 can represent our code generation technique for *Alternative* connection part. As it can be seen both in class diagram in Figure 4.9 and in generated source code pieces in Figure 4.10, our generator adds both a list, namely, "featureAList", of "IFeatureA" type and a field with a name called "selectedFeatureA" into the "FeatureA" class as attributes. The field of "selectedFeatureA" represents the feature that will be selected from between the alternatives in the configuration file. Our configurator adds also getter and setter methods of private attributes in the generated class during all generation process. The aim of this mechanism is to provide the alternative subfeatures, which always uses similar functions but having different implementations, to use same interfaces in order to generate an easily configurable object oriented design for variability management. That is, our code generator checks the product configuration file and determine the selected feature from the alternative subfeatures and then, set the "selectedFeatureA" attribute of the instance of "FeatureA" as a new instance of this selected feature like the source code in Figure 4.10.

4.4.6 *Or* Connection

The other connection type is *Or* that means the subfeatures of a concrete *Or* feature can be selected multiple in the product configuration. This connection in Figure 4.11 resembles *Alternative* connection. The result UML class diagram of such a feature model in Figure 4.11 is also same as the *Alternative* connection of features in Figure 4.9.

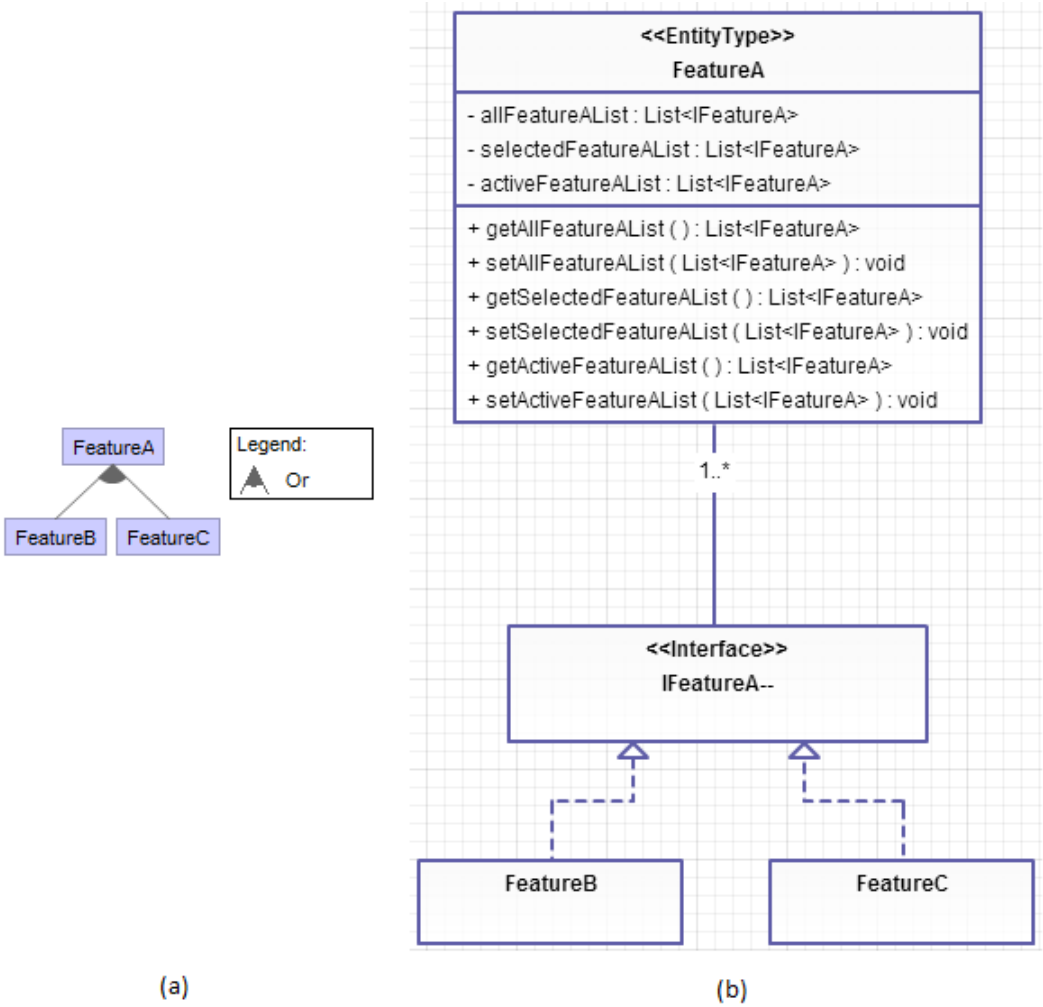

Figure 4.11: *Or* Connection Features (a) and The Corresponding UML Class Diagram (b)

The main difference of the two connections is the number of features that can be selectable in a product configuration. Only one child feature of an *Alternative* parent feature can be selected whereas one child feature or more child features of *Or* parent

feature can be selected in product configuration. Therefore, data structures generated for *Or* feature is a little different for *Alternative*. In code generation phase for *Or* connection, the generator creates firstly an entity class of concrete "FeatureA" without fields and methods and then, creates a type called "IFeatureA" by using the parent name. After the generation of this type that is used to provide a common interface for child features, the entity classes of concrete subfeatures "FeatureB" and "FeatureC" are generated. Our generator makes the classes of subfeatures implement the interface generated with the type of "IFeatureA" just before these classes are created. The UML Class Diagram of generated source code in Figure 4.12 for this feature model in Figure 4.11 can represent our code generation technique for *Or* connection part. As it can be seen in both class diagram in Figure 4.11 and generated source code templates in Figure 4.12, the generator adds a list field called "allFeatureAList", a list field called "selectedFeatureAList" and a list field called "activeFeatureAList" with the type of "IFeatureA" into the "FeatureA" class as attributes. However, if "FeatureA" feature were an *Alternative* feature, a field called "selectedFeatuerA" with type "IFeatureA" would insert into the "FeatureA" class instead of the list field and the field of "activeFeatureAList" would not be inserted. The field of "selectedFeatureAList" indicates the multiple features that will be selected from the subfeatures in the configuration file. In run-time, the configurator adds the selected features in configuration into this field. The aim of this *Or* mechanism is to provide the multiple-selectable subfeatures, which always uses similar functionalities but having different implementations, to use same interface types in order to generate an easily configurable object oriented design for variability management. That is, our code generator checks the product configuration file and determine the selected features from the these subfeatures and then, add a new instance of each selected subfeature to "selectedFeatureAList" attribute of the instance of "FeatureA", such as the source code in Figure 4.12. The configurator inserts also getter and setter methods of the above private attributes into the generated classes during all generation process.

4.4.7 Abstract Feature

Two main type of features called *Abstract* and *Concrete* feature are used in the feature modeling in FeatureIDE. The properties of *Concrete* features are handled in above

```
public interface IFeatureA
{

}

public class FeatureB
    implements IFeatureA
{

}

public class FeatureC
    implements IFeatureA
{

}
```

```
public class FeatureA {
    private List<IFeatureA> allFeatureAList;
    private List<IFeatureA> selectedFeatureAList;
    private List<IFeatureA> activeFeatureAList;

    public FeatureA() {
        this.setAllFeatureAList(
                new ArrayList<IFeatureA>());
        this.setSelectedFeatureAList(
                new ArrayList<IFeatureA>());
        this.setActiveFeatureAList(
                new ArrayList<IFeatureA>());
        this.getAllFeatureAList().add(
                new FeatureB());
        this.getAllFeatureAList().add(
                new FeatureC());
    }

    public List<IFeatureA> getAllFeatureAList() {
        return this.allFeatureAList;
    }
    public void setAllFeatureAList(
            List<IFeatureA> allFeatureAList) {
        this.allFeatureAList = allFeatureAList;
    }
    public List<IFeatureA> getSelectedFeatureAList() {
        return this.selectedFeatureAList;
    }
    public void setSelectedFeatureAList(
            List<IFeatureA> selectedFeatureAList) {
        this.selectedFeatureAList = selectedFeatureAList;
    }
    public List<IFeatureA> getActiveFeatureAList() {
        return this.activeFeatureAList;
    }
    public void setActiveFeatureAList(
            List<IFeatureA> activeFeatureAList) {
        this.activeFeatureAList = activeFeatureAList;
    }
```

Figure 4.12: Generated Source Code for *Or* Connection

sections. There are different ideas about what an abstract feature and how it can be used. Thomas et al. has proposed that an abstract feature used for structuring a feature model but it does not affect the implementation part with respect to analysis about different product line techniques [33].

In the light of this information, we decide that an abstract feature has no mapping to an entity class during our code generation process and it can be used to create a type in order to structure the generated source code templates. For example, the abstract feature called "FeatureB" of the feature model in Figure 4.13 has an *Alternative* connection type. We assumed that this abstract feature is used just an abstraction to structure the feature model. It implies that "FeatureC" and "FeatureD" have common functionalities but different implementations and it is allowed to be selected just one

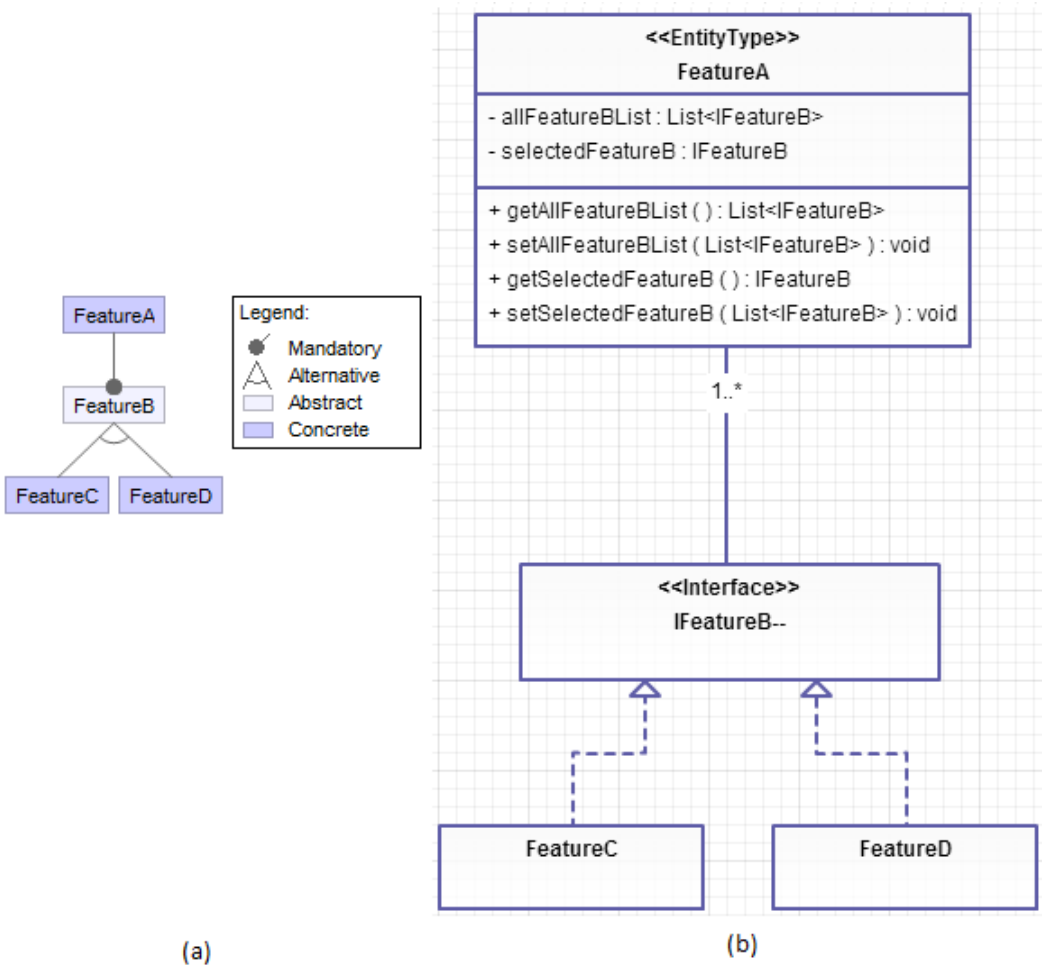

Figure 4.13: *Abstract* Feature (a) and The Corresponding UML Class Diagram (b)

of them in product configuration file. The UML class diagram created by using the feature model in Figure 4.13 as an input shows that the name of abstract feature is just used to generate a common type for alternative features called "FeatureC" and "FeatuerD". However, the abstract feature does not any mapping with implementation part and so, the generator does not create any class for it. In code generation process, the generated source code templates for the model in Figure 4.13 resembles the templates generated for concrete *Alternative* feature in Figure 4.10. We generated code templates and data structures as if the abstract feature, "FeatureB" was not in the feature model. That is, the feature model in Figure 4.13 can be assumed as the feature model in Figure 4.9, but the only difference between the two model is that the list fields added into the class of "FeatureA" has the type called "IFeatureB", which is determined by using the name of the abstract feature, instead of "IFeatureA".

4.4.8 Configuration Classes

While source code templates are created by our generator, an *Application* java class that has source code pieces to be able to start up the final application by using the product configuration file including the selection of features and a *Configuration* java class that takes the textual list of ordered features in the selected product configuration as input and operates with each features in the list in order to make bindings in run-time with the data structures added into the generated entity classes are developed completely and automatically by our generator. The *Configuration* class is created as an empty class at the beginning of code generation process.

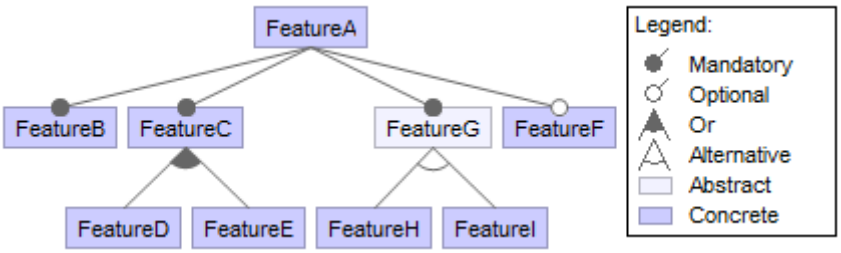

Figure 4.14: An Example Feature Model

The generator constitutes a conditional code statement according to the characteristics of current feature and its parent feature and then, it is added into the class file, while the tree structure of a feature model is being parsed recursively. For each feature in the feature model except abstract features, the content of this statement is specified by analyzing the parent features recursively from bottom to up. Any code statement is not added into this configurator class for abstract features. They are just used for specifying a name of type to structure the architecture of the generated code templates. In order to show how the configurator class is created and how it configures the final application in run-time, development of an example application is explained step by step. Firstly, a feature model in Figure 4.14 is designed to cover the various feature types and connections mentioned in this chapter. The model covers *Mandatory* features, an *Optional* feature, an *And* root feature, an *Or* connection and an *Alternative* connection with an abstract parent feature. Our code generator creates the *Configuration* java class with a method called "executeFeature" that takes a string for feature name and an instance of the class of root feature as inputs and code templates with data structures represented by an example UML class diagram in Figure

4.15 after it processes this model. Then, software developers continue implementing the application on the generated templates and they use the data structures in these templates according to program variants. When the application is completed and run by *Application* class created by our generator, the product configuration file including feature selections in order is send to an instance of *Configuration* class as an input by *Application* class. The instance of *Configuration* class checks each name of feature in the configuration file in order and then, makes required settings for each feature in the method of "executeFeature" in Figure 4.15. The method provides application to be able to work for all feature combinations taken as an input.

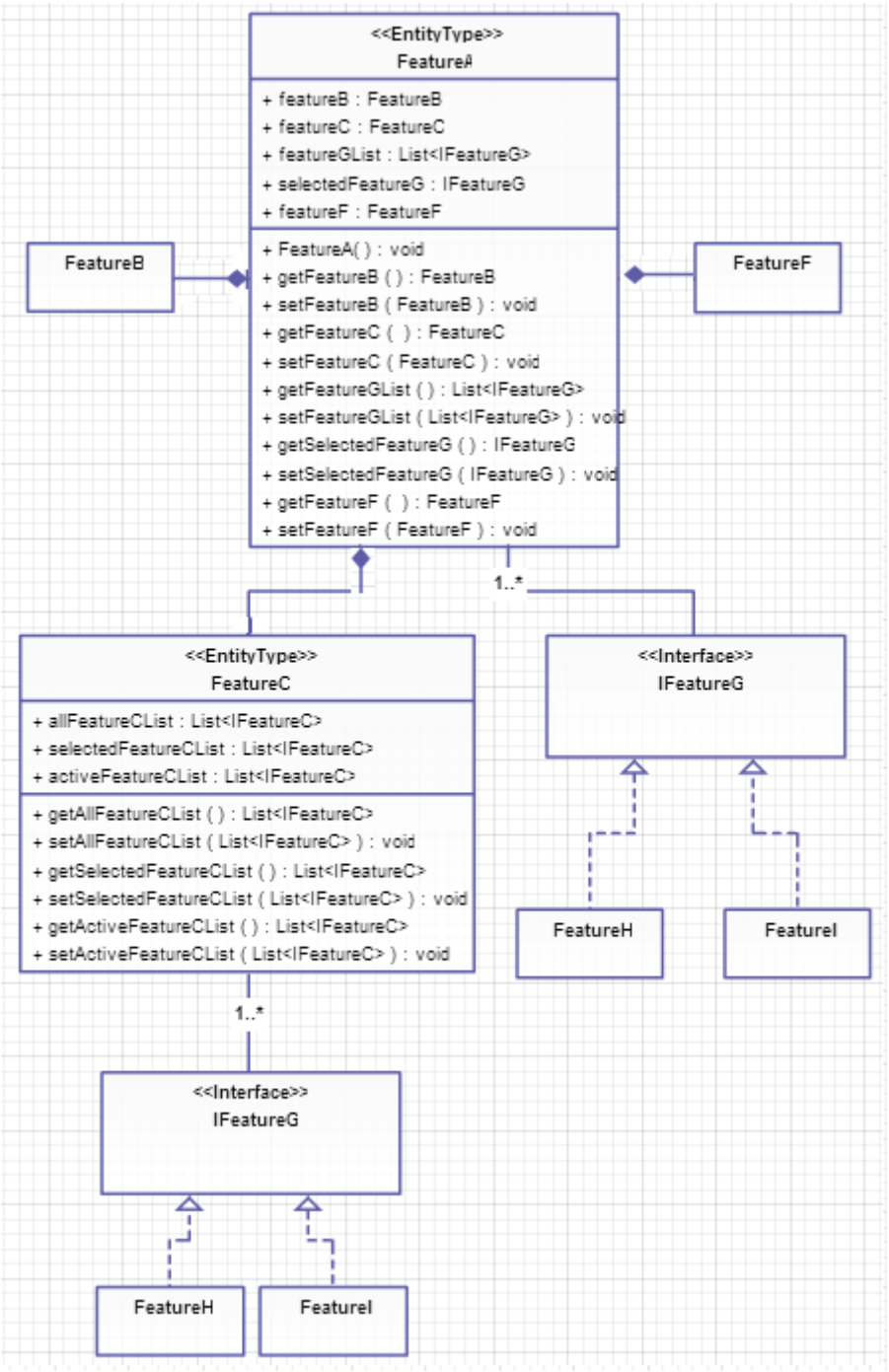

Figure 4.15: An Example UML Class Diagram

CHAPTER 5

CASE STUDIES IN ONLINE SHOPPING DOMAIN

Two case studies performed to give an example for the proposal made in Chapter 4 is showed with all details in this chapter. In the first case study, a synthetic feature model is examined in order to show generation of code templates for high-level feature models including all feature connection types. For the second case study, we select online shopping domain as an example domain for our work in order to implement our approach and then we select FeatureIDE[32] as feature modeling tool. In this section, the workflow of developing E-Shop software system by using this approach is described as follows: domain analysis of online shopping, the feature model specified for a product family of E-Shop System, identifying various product configurations, UML class diagram for the feature model, class templates, generating source code templates and finally E-Shop application.

5.1 A Synthetic Model

The overflow of our approach was detailed in Chapter 5. As it can be seen in this chapter, we have an algorithm to understand the tree structure of a feature model including design decisions and requirements in a domain and then it generates source code templates by using the information in the feature model. For example, *Mandatory* and *Optional* features, *Alternative*, *And* and *Or* connections between a parent feature and its child features, *Abstract* and *Concrete* features. A feature model has also the information of cross-tree constraints between features. All these information helps the algorithm to specify data structures in entity classes for each case in feature

model.

In order to test our template code generation algorithm, we decided to create a feature model containing all these connection types between features and all feature types in feature modeling language that we used in our approach. Then, a synthetic feature model in Figure 5.1 is constituted and so, we try to put various cases in this model in order to force the limits of our generation technique.

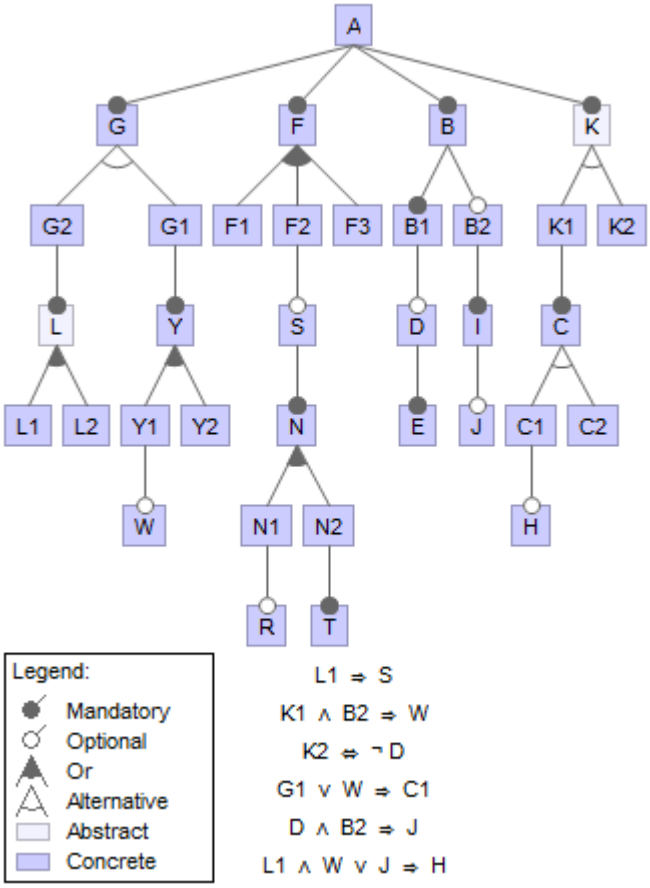

Figure 5.1: A Synthetic Model in FeatureIDE

This feature model aims to test our algorithm whether it can generate correctly and completely source code templates or not. However, it does not represent any domain and it includes features with the names of single letters. When we operate our algorithm on this model, it can successfully create all types, interfaces, entity classes with all correct data structures and getter and setter methods. In addition to them, *Configuration* class that will take the textual list of ordered features in the selected product configuration as input and operate with each features in the list in order to make bindings in run-time with the data structures added into the generated entity classes are

developed completely and automatically by our generator. The UML class diagram that demonstrates the code templates can be seen in Figure 5.2.

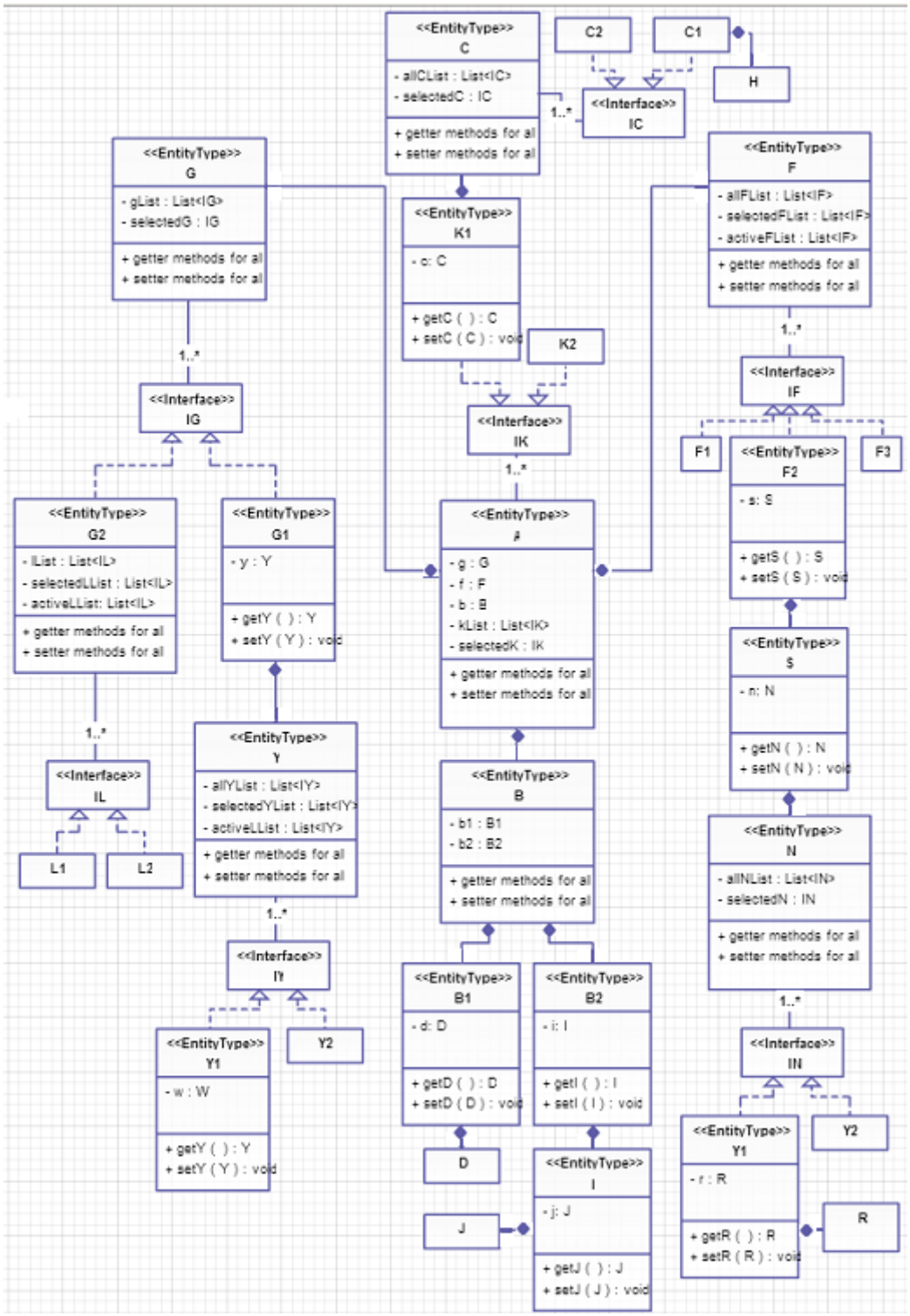

Figure 5.2: UML Class Diagram for The Synthetic Model

This class diagram demonstrates mappings between code templates and all connec-

tions between features in the synthetic model. This shows also that our algorithm can generate correct results for a complex and large feature model.

5.2 Online Shopping Domain

Figure 5.3: Online Shopping Principle

Online shopping or e-shopping means the act of purchasing products or services over the Internet. It can be defined as a form electronic commerce which allows consumers to directly buy goods or services from a seller over the Internet using a web browser. Recently, many corporations such as Amazon.com, eBay etc. become popular all over the world rather than physical stores. Software companies have begun to develop online shopping software systems by offering online store interfaces for consumers.

We tried to develop a desktop shopping application instead of a web-based application so that the performing our approach on this case study can be demonstrated more easily. Thus, the limits of the domain are approximately specified. The next step is to specify requirements in this domain and then the commonalities and variabilities in the domain can be determined by domain engineers. Customers may want various features for a shopping application. When the commonality and variability points are captured successfully, only one application that can be work for all combinations of that domain can be developed by this approach instead of developing a different

application for each domain decision selections.

As indicated in above, the design choices of the domain is needed to be specified first of all. A shopping application is basically composed of a product pool, the store interface, a security system and a payment system. Moreover, a property of providing a catalogue for products in the system and a component for campaigns in the online market can be added to the domain choices as mandatory. A search tool also can be added as an optional choice. In order to structure and organize the connections and constraints between the design choices, a feature model will be designed in the next section.

5.2.1 Feature Model for E-Shop System

An example feature model is designed according to the design choices specified in the previous section and it is presented in Figure 5.4.

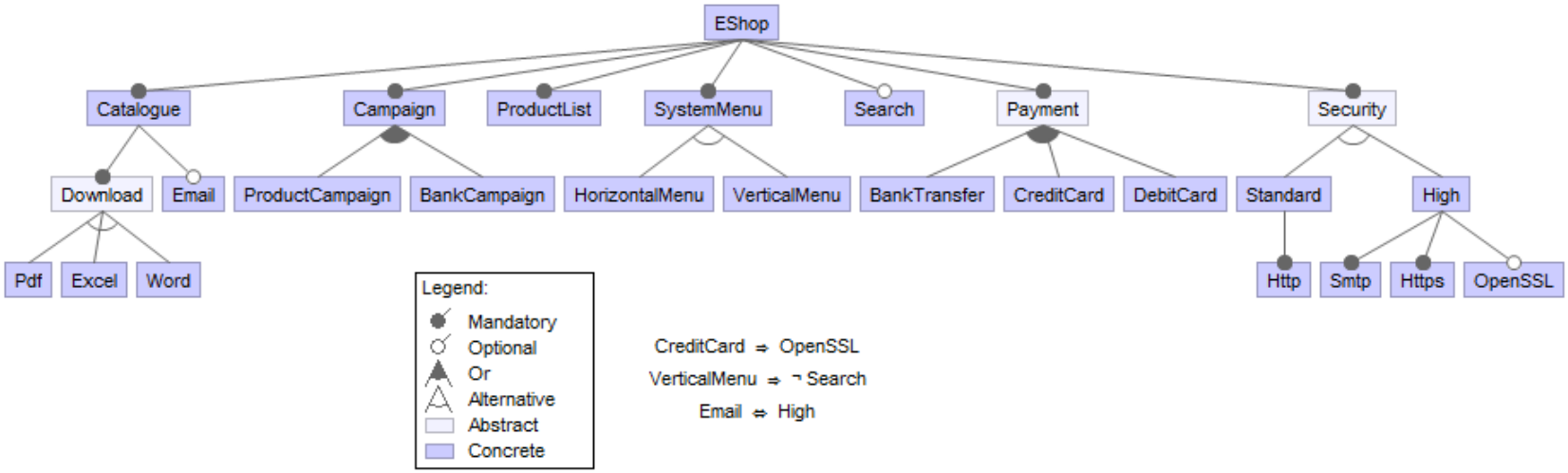

Figure 5.4: E-Shop Feature Model

First of all, to express the commonality and variability points in the feature model can facilitate the following steps of our work.

- *EShop* represents the root feature of this feature model. This feature is the main part of the model and it is accepted as a start point of the model.
- *Catalogue* and its subfeatures denote the design of the catalogue of products in the market. *Catalogue* is designed to have two child features in order to separate the types of catalogue support for users. The first child feature is *Download* that is an abstract feature as it can be seen in Figure5.4. The abstract feature is just used to structure its subfeatures. That is, *Download* is used to create a common type called for the subfeatures and it does not have any mapping with implementation artifacts. *Pdf*, *Excel* and *Word* features represents the alternative download formats and they are designed as *Alternative* connection type with their parent feature *Download*. The second child feature of *Download* is *Email* represents the ability of sending e-mail of the catalogue as an optional feature in the model.
- *Campaign* feature represents the campaign management of the system and this features serves two main mechanism for campaigns called *ProductCampaign* and *BankCampaign*. The two choices can be selected multiple in the configuration due to the logic of *Or* connection type between them and their parent feature. *BankCampaign* symbolizes the campaigns for various credit cards and banks. On the other hand, *ProductCampaign* symbolizes the campaigns for the products in the market such as sale campaigns.
- *ProductList* is a mandatory feature since it represents the product pool of the market. This feature must be in all configurations of this system.
- *SystemMenu* is a mandatory feature that have an *Alternative* connection with *HorizontalMenu* and *VerticalMenu* features. This compound feature represents the store interface of the system and the subfeatures provide selectable interfaces for shopping.
- *Search* feature denotes a basic search tool to search the products recorded in the system and show the details of them.

- *Payment* is an *Abstract* feature to organize the its three subfeatures that each one represents a different kind of payment system. This payment system can be selected multiple in the configuration process and they can be used at the same time in run-time of the application. The first child feature, *BankTransfer* symbolizes the direct transfer of the payment from bank, while *CreditCard* represents the payment type with credit cards. The third choice is *DebitCard* that provides the cardholder electronic access to his or her bank account.

- *Security* is also an *Abstract* feature that provide alternative security system for the whole system with its subfeatures. The child feature called *Standard* is a standard security system including *Http* feature representing Hypertext Transfer Protocol to provide data communication. The other child feature called *High* is a high security system containing *Smtp*, a simple mail transfer protocol, and *Https*, Secure Hypertext Transfer Protocol, as mandatory features. Moreover, an optional subfeature called *OpenSSL* represents a security aspect with an open-source implementation of the SSL and TLS protocols.

After the definitions of features in the model, the constraints included in the model can be explained as follows:

- The first constraint is "CreditCard => OpenSSL" means that when *CreditCard* is selected as a payment type, *OpenSSL* security aspect below *High* feature is automatically selected in configuration.

- The second constraint "VerticalMenu *implies* not Search" represents that when *VerticalMenu* is selected as a menu type, search tool is directly inactive.

- "Email <=> High" constraint denotes that *Email* feature can be selected if *High* feature is selected and *High* feature can be selected if *Email* feature is selected.

5.2.2 UML Class Diagram for E-Shop Feature Model

We mentioned that a code generation algorithm was implemented to generate source code templates containing interfaces, classes with data structure patterns to be used

for managing variability. The UML class diagram of generated class templates from E-Shop feature model can be seen in detail in Figure 5.5.

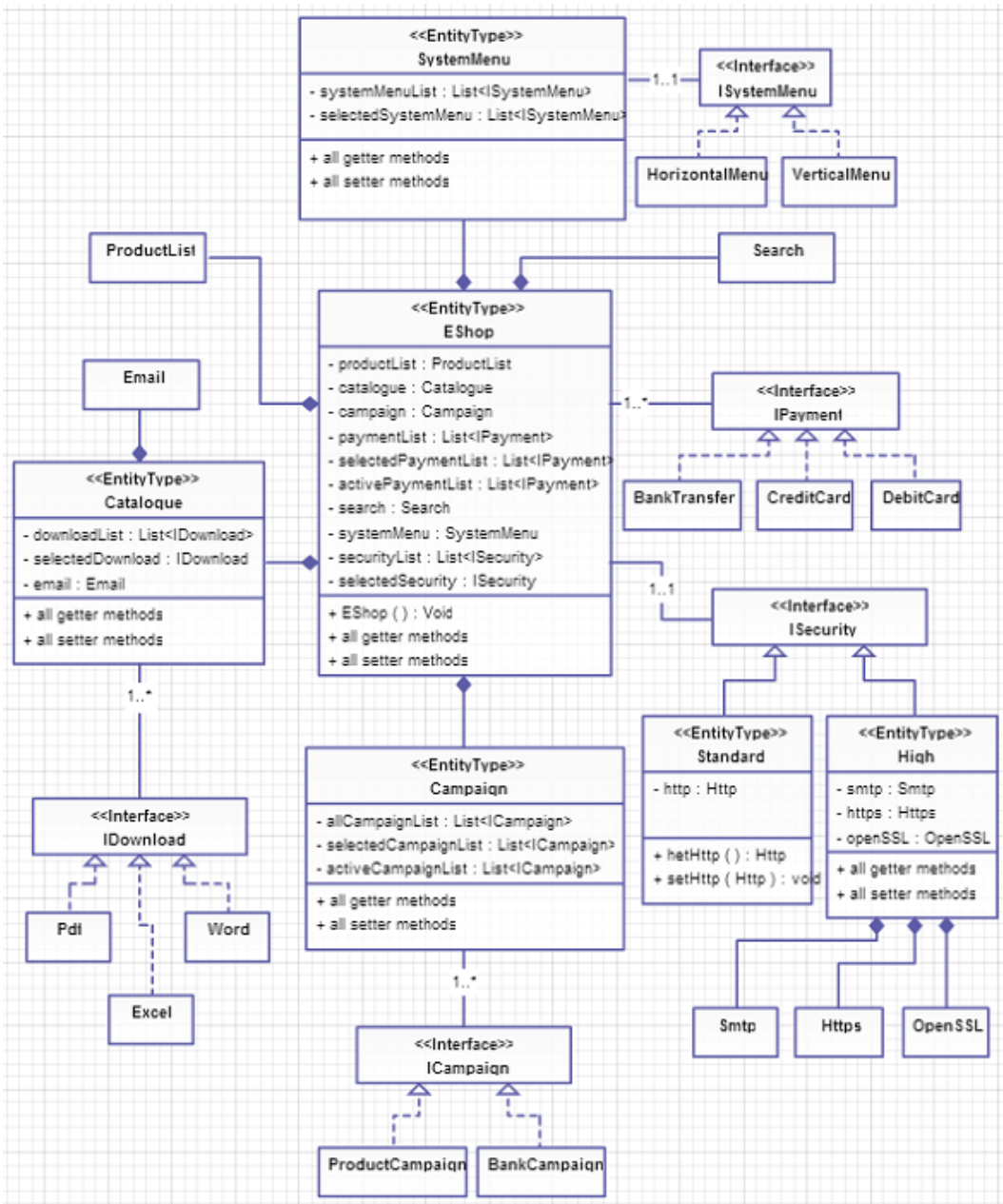

Figure 5.5: UML Class Diagram of E-Shop System

5.2.3 Generation of Class Templates

The code templates of corresponding the relationships and feature types in feature model are listed below according to Figure 5.5:

- *And*: *EShop* is a mandatory *And* feature mapped to "EShop" entity class and this class is composed from *Catalogue*, *Campaign*, *ProductList*, *SystemMenu* and *Search* classes. This class includes also a list with the type of "IPayment" and a list with the type of "ISecurity".

- *Mandatory*: *ProductList* is a mandatory feature and it is mapped to "ProductList" entity class and it is added into "EShop" class as a field with getter and setter methods.

- *Optional*: *Search* is an optional feature and it is mapped to "Search" entity class in code generation process. Then, it is added into the "EShop" class as a field with getter and setter methods. *Optional* connection is as same as *Mandatory* connection for code generation part of this study. However, the difference between them can be seen when they are captured during binding process in-runtime.

- *Alternative*: *SystemMenu* has an alternative connection with its subfeatures. "SystemMenu" entity class is generated firstly and then, "HorizontalMenu" and "VerticalMenu" entity classes implementing "ISystemMenu" interface are generated in this part. Moreover, our generator adds both a list, namely, "systemMenuList", and a field with a name called "selectedSystemMenu" of "ISystemMenu" type into the "SystemMenu" class as attributes. The "systemMenuList" will contain all the alternatives and "selectedSystemMenu" will contain just the selected feature in the configuration in run-time.

- *Or*: *Campaign* has an *Or* connection with its subfeatures. "Campaign" entity class is generated firstly and then, "ProductCampaign" and "BankCampaign" entity classes implementing "ICampaign" interface are created for this connection. Furthermore, our generator adds "allCampaignList", "selectedCampaignList" and "activeCampaignList" into "Campaign" class as a list field with the type of "ICampaign" separately.

- *Abstract - Alternative*: *Security* has the properties of both *Abstract* and *Alternative*. For this connection type, any entity class is not generated as if this feature is not in feature model. Then, a list called "securityList", and an attribute called

"selectedSecurity" with the type of "ISecurity" are added into "EShop" entity class like in the above *Alternative* connection.

- *Abstract - Or*: *Payment* has the properties of both *Abstract* and *Or*. For this connection type, again any entity class is not generated and the three lists with the type of "IPayment" are added into "EShop" entity class like for *Or* connection as if "Payment" is not in the feature model. Abstract features are just used to structure the model.
- *Abstract - And*: Since this type of connection is meaningless, we ignore this case and it is assumed not to be used.

In addition to the classes, a *Configuration* class is created and it is evolved with each feature in the feature model by adding some code statements that will make bindings in the application during run-time. That is, this class will take a product configuration file as an input in run-time to activate selected features for application and deactivate the features not in the configuration so that this automatically generated class provide application manage automatically the variabilities in the domain without a manual setting during run-time process.

5.2.4 E-Shop Application

After the code generation process is ended, developers continue development of E-Shop application on generated template classes and within data structures as if this application had the ability of all features in feature model. The consideration of that all features can be active in application is based on the whole development process. After development process, various configuration files can be created by selecting a set of features from E-Shop feature model. In the next section product configuration examples are given.

5.2.5 Possible Combinations of E-Shop Feature Model

After domain analysis and the development process are finished, a valid product configuration is designated by selecting features according to the rules and constraints

in the feature model. A valid configuration is needed to run the E-Shop application so that the application can take the configuration as an input and then it can operate this by using *Configuration* class generated automatically. On contrary, the application does not work for an invalid application. FeatureIDE tool is used to determine whether a configuration is valid or invalid by using all rules and cross-tree constraints for the feature model. Moreover, this tool serves a simple interface to make selections easily. An example configuration in Figure 5.6 is decided as invalid by FeatureIDE because there are mandatory selections to be made and constraints to be obeyed in the configuration.

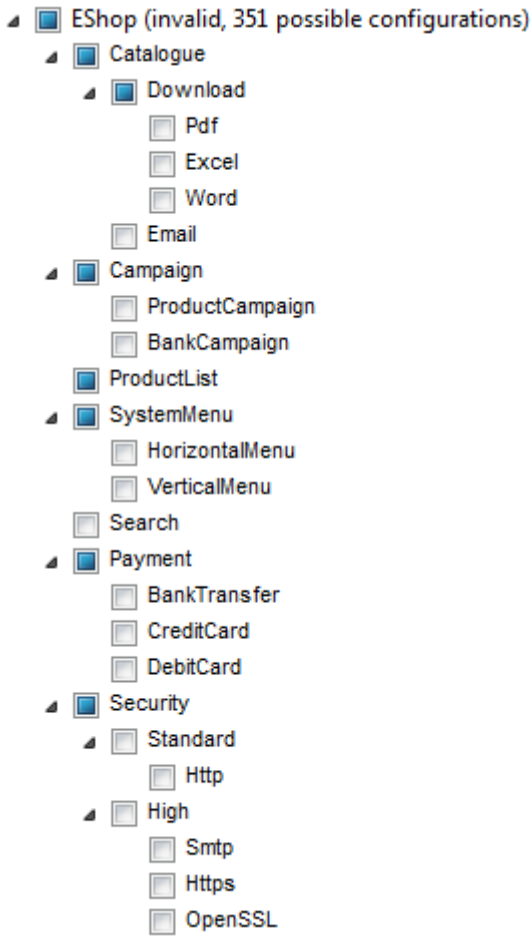

Figure 5.6: An Invalid Configuration of E-Shop System

When a suitable set of features is selected, FeatureIDE shows that the configuration becomes valid like in Figure 5.7. After a valid configuration is specified, the final work is just run the application. The complete application is designed automatically to take the valid configuration as an input and then it is send to the *Configuration*

instance. Then, this instance checks the set of selected features and makes the required bindings automatically in run-time process so that the application can work for the selected features. When various feature combinations are send as an input, the application developed by our approach can work for all feature combinations in the domain and the selections of the domain can be reflected directly and automatically to the application.

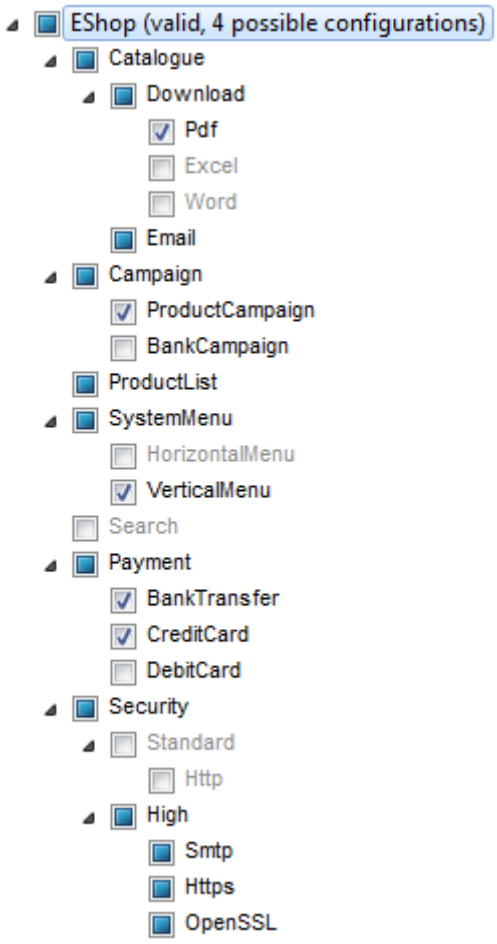

Figure 5.7: An Invalid Configuration of E-Shop System

CHAPTER 6

CONCLUSION

In this chapter, a summary of our approach is explained in the first section. Then, the advantages and disadvantages of the results of our case study are given in Discussion Section. Finally, what can be done in future work is described.

6.1 Summary of Conducted Approach

An approach to manage variability in object-oriented applications by using feature modeling is proposed in this thesis. The most critical part of this work is how to use the variability information to manage the different configurations of applications. The code generation patterns are specified to generate source code templates with data structures and complete configurator classes that will be used to manage variability of the domain by using the feature selections. Then, the data structures in template codes help software developers to continue practically development of the application by using the architecture provided. After the development phase is completed, the configurator classes created automatically by our generator use both the configuration file including design decisions and data structures generated in the beginning of this work to make bindings and manage variability in run-time of the application.

For feature modeling language, FeatureIDE tool that provides a straightforward interface for creating feature models and defining constraints between features is used in our work. This tool also has a configurator to provide user to select features from the final feature model according to the rules and constraints specified for domain and then, a configuration file containing the selected features in order is created. In order

to show the details of our approach, a case study is implemented for an online shopping domain. The case study in Chapter 5 gives a detailed overview of the flow of our approach. In this study, how our approach can be applied easily on a development of an application and how the application can work for each set of features in E-Shop domain is shown step by step.

6.2 Discussion

There are some points to be discussed to describe the positive and negative aspects of this study.

First of all, this study aims to build a model driven architecture. At the point of producing code from modeling diagrams, just code templates are generated by our code generator technique. However, the code pieces that are close to the final state of a software system would be generated if more extended feature modeling language than FeatureIDE were used. Moreover, another disadvantage of our approach may be reverse engineering in model driven engineering field. In this study, design decisions in a domain prepared by domain engineers during domain analysis are supposed to be well defined and formed into a feature model. The feature model must represent commonalities and variabilities in the domain and so the development of application can progress properly and the final application can manage automatically the variabilities in the domain. If the management issue is not handled carefully, the problems may be occurred in the next steps of the approach.

The second point is the code generation process in this approach. Our generator creates the code templates including special data structures by mapping to the feature model. The code generation process can be extended by using a different feature modeling technique. The feature modeling of FeatureIDE used in our approach represents the information of names, types and connections of features in the model. A feature in this model does not have the information of functionalities and attributes that are needed to be in implementation level. If such a feature modeling language that includes both the information in FeatureIDE and the extra information of attributes and functionalities of features is implemented, the code generation process

of our approach can be improved and so developers can complete the development application in a short time period. However, the feature modeling in FeatureIDE is more comprehensible than a feature modeling that will include more information. Moreover, developers do not need to learn any extra language in order to apply our approach because knowledge of object oriented programming is enough to implement this technique. This point is a very important advantage in terms of time saving for software development.

Another point is about developing the application. The generated template codes by our generator form a basis in order to begin to the development. Developers can add new classes or new functionalities to the application as long as the changes do not affect the variabilities of the domain. However, the data structures inserted into these template classes in code generation process by using our variability patterns must be stable during all development time. This issue is the most critical part of our approach. When developers keep and used all data structures in the templates during the implementation of the application, all variabilities in the domain can be reflected automatically into the final application through the selected features in the feature model.

6.3 Future Work

For future work, large-scale domains are used to try to apply our approach, so that the insufficient aspects of this approach can be determined for such domains. Then, it can be improved in order to develop applications for large-scale markets and companies. Furthermore, FeatureIDE tool used in our approach is an open-source an extensible framework for feature oriented software development based on Eclipse. An extension using our approach can be added to this framework in order to follow easily the steps of our approach as a future work. This extension includes also a function to trace variability of features. For instance, where and how a feature is used in the source code can be traceable. Moreover, a mechanism for reverse engineering can be built by using this traceability. And finally, our approach may be extended to be applied on applications using various programming languages and techniques in addition to object-oriented applications.

REFERENCES

[1] Codemodel api. from https://codemodel.java.net/nonav/apidocs/com/sun/codemodel/package-summary.html, 2014. "last accessed on 18/08/2014".

[2] Domainengineering. from http://en.wikipedia.org/wiki/Domain_engineering, 2014. "last accessed on 12/08/2014".

[3] Xfeature modeling tool. from http://www.pnp-software.com/XFeature/, May 2014. "last accessed on 12/08/2014".

[4] Sven Apel and Christian Kästner. An overview of feature-oriented software development, 2009.

[5] Sven Apel, Thomas Leich, Marko Rosenmüller, and Gunter Saake. Featurec++: On the symbiosis of feature-oriented and aspect-oriented programming. In *In Proceedings of the International Conference on Generative Programming and Component Engineering*, pages 125–140. Springer, 2005.

[6] Sven Apel, Christian Lengauer, Bernhard Möller, and Christian Kästner. An algebra for features and feature composition. In *Proceedings of the 12th International Conference on Algebraic Methodology and Software Technology (AMAST)*, volume 5140 of *Lecture Notes in Computer Science*, pages 36–50, Berlin/Heidelberg, Germany, July 2008. Springer-Verlag. Acceptance rate: 47 % (27/58).

[7] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Trans. Softw. Eng.*, 30(6):355–371, June 2004.

[8] Don Batory. Feature models, grammars, and propositional formulas. In *Proceedings of the 9th International Conference on Software Product Lines*, SPLC'05, pages 7–20, Berlin, Heidelberg, 2005. Springer-Verlag.

[9] Don Batory. A tutorial on feature oriented programming and the ahead tool suite. In *Proceedings of the 2005 International Conference on Generative and Transformational Techniques in Software Engineering*, GTTSE'05, pages 3–35, Berlin, Heidelberg, 2006. Springer-Verlag.

[10] Gamze Bayraktar. Representing component variability in configuration management. Master's thesis, Middle East Technical University, 2012.

[11] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.*, 35(6):615–636, September 2010.

[12] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated reasoning on feature models. In *Proceedings of the 17th International Conference on Advanced Information Systems Engineering*, CAiSE'05, pages 491–503, Berlin, Heidelberg, 2005. Springer-Verlag.

[13] Jan Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[14] Andreas Classen, Patrick Heymans, and Pierre-Yves Schobbens. What's in a feature: A requirements engineering perspective. In *Proceedings of the Theory and Practice of Software, 11th International Conference on Fundamental Approaches to Software Engineering*, FASE'08/ETAPS'08, pages 16–30, Berlin, Heidelberg, 2008. Springer-Verlag.

[15] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged configuration using feature models. In *Software Product Lines: Third International Conference, SPLC 2004*, pages 266–283. Springer-Verlag, 2004.

[16] Ali H. Dogru and Murat M. Tanik. A process model for component-oriented software engineering. *IEEE Softw.*, 20(2):34–41, March 2003.

[17] William B. Frakes and Kyo Kang. Software reuse research: Status and future. *IEEE Trans. Softw. Eng.*, 31(7):529–536, July 2005.

[18] Guozheng Ge. *Rhizome: A Feature Modeling and Generation Platform for Software Product Lines*. PhD thesis, Santa Cruz, CA, USA, 2008. AAI3338586.

[19] Hassan Gomaa. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[20] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[21] Kyo C Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. Form: A feature-; oriented reuse method with domain-; specific reference architectures. *Annals of Software Engineering*, 5(1):143–168, 1998.

[22] Christian Kästner. Virtual separation of concerns: Toward preprocessors 2.0, 5 2010. Logos Verlag Berlin, isbn 978-3-8325-2527-9.

[23] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, UK, 2001. Springer-Verlag.

[24] Kwanwoo Lee, Kyo Chul Kang, Wonsuk Chae, and Byoung Wook Choi. Feature-based approach to object-oriented engineering of applications for reuse. *Softw., Pract. Exper.*, 30(9):1025–1046, 2000.

[25] Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[26] A. Pasetti and O. Rohlik. The XFeature project: Development of a Product Family Modeling Tool. Technical report, July 2005.

[27] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[28] Christian Prehofer. Feature-oriented programming: A fresh look at objects. pages 419–443. Springer, 1997.

[29] Ina Schaefer, Lorenzo Bettini, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond*, SPLC'10, pages 77–91, Berlin, Heidelberg, 2010. Springer-Verlag.

[30] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. A taxonomy of variability realization techniques: Research articles. *Softw. Pract. Exper.*, 35(8):705–754, July 2005.

[31] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. Featureide: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 79:70–85, 2014.

[32] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. Featureide: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 79:70–85, 2014.

[33] Thomas Thum, Christian Kastner, Sebastian Erdweg, and Norbert Siegmund. Abstract features in feature modeling. In *Proceedings of the 2011 15th International Software Product Line Conference*, SPLC '11, pages 191–200, Washington, DC, USA, 2011. IEEE Computer Society.

[34] Pamela Zave. Programming methodology. chapter An Experiment in Feature Engineering, pages 353–377. Springer-Verlag New York, Inc., New York, NY, USA, 2003.

[35] Hongyu Zhang and Stan Jarzabek. Xvcl: A mechanism for handling variants in software product lines. *Sci. Comput. Program.*, 53(3):381–407, December 2004.

[36] Tewfik Ziadi, Loïc Hélouët, and Jean marc Jézéquel. Towards a uml profile for software product lines. In *In PFE*, pages 129–139. Springer, 2003.