A Textual Software Product Lines Design Model By Mixing Class and Feature Concepts

By

Ola Abdel Raoof Younis

Supervisor
Prof. Said Ghoul

This Thesis was Submitted in Partial Fulfilment of the Requirements for the Master's Degree in Computer Science

Deanship of Academic Research and Graduate Studies
Philadelphia University

2013

جامعة فيلادلفيا
نموذج تفويض

أنا علا عبد الرؤوف سليمان يونس ، أفوض جامعة فيلادلفيا بتزويد نسخ من رسالتي للمكتبات أو المؤسسات أو الهيئات أو الأشخاص عند طلبها.

التوقيع :
التاريخ :

Philadelphia University
Authorization Form

I am, Ola Abdel Raoof Suliman Younis, authorize Philadelphia University to supply copies of my thesis to libraries or establishments or individuals upon request.

Signature:

Date:

A Textual Software Product Lines Design Model Mixing Class and Feature Concepts

By

Ola Abdel Raoof Younis

Supervisor
Prof. Said Ghoul

This Thesis was Submitted in Partial Fulfilment of the Requirements for the Master's Degree in Computer Science

Deanship of Academic Research and Graduate Studies
Philadelphia University

2013

Successfully defended and approved on _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

Examination Committee Signature		**Signature**
Dr. Academic Rank:	, Chairman.	_ _ _ _ _ _ _ _ _ _ _ _ _
Dr. Academic Rank:	, Member.	_ _ _ _ _ _ _ _ _ _ _ _ _
Dr. Academic Rank:	, Member.	_ _ _ _ _ _ _ _ _ _ _ _ _
Dr. Academic Rank:	, External Member.	_ _ _ _ _ _ _ _ _ _ _ _ _

Dedication

I dedicate this work to my husband *Dr. Mohammad Alomari,*
Who encouraged me all the way to reach this point, with his personal support, great patience at all times, and his endless love and support….

Ola A. Younis

Acknowledgment

It would not have been possible to write this master thesis without the help and support of the kind people around me, to only some of whom it is possible to give particular mention here.

Above all, I would like to express my thanks and sincere gratitude for who has guided me through my study and my thesis work; my supervisor *prof. Said Ghoul*, for giving the wisdom, strength, support and knowledge in exploring things.

I would like to thank my family members; Parents, brothers, sisters and my children *Omar* and *Aya* for giving me their unequivocal support throughout, as always, for which my mere expression of thanks likewise does not suffice.

Also, I am grateful for those who supported me and encouraged me in any way; my teachers at Philadelphia University, my friends and my superiors and colleagues at work.

Ola A.Younis

Table of Contents

List of Tables

List of Abbreviations

Abbreviation	Full Name
SPL	Software product line
DSL	Domain specific language
OO	Object oriented
FTR	Features_Types relation
FTF	Features_types feature
GR	Global relation
GF	Global feature
CR	Control relation
ConR	Configuration relation
ConF	Configuration feature
OOPL	Object oriented programming language
TDM	Textual SPL design methodology

List of Figures

Abstract

Designing software product line (SPL) is very important for increasing system reusability and decreasing cost and efforts for building components from scratch for each software configuration.

Several approaches handled SPL engineering process with several techniques. The most famous one was done by separating the commonalities and variability for system's components to allow configuration selection based on user defined features. These approaches deal with all software development phases, but the challenge and important phases are design and implementation.

Textual notation-based approaches have been used for their formal syntax and semantics to represent system features and implementations. But these approaches are still weak in the mixing features (conceptual level) and classes (physical level) that guarantee smooth and automatic configuration generation for software releases.

In this thesis, we will enhance SPL process by defining meta-features that captures the most important characteristics of feature modelling concepts, and classifying these features according to their functionalities. We will allow mixing class and feature concepts in a simple way using class interfaces and inherent features for smooth move from feature model to class model.

SPL process will be enriching with a textual design and implementation methodology mixing class and feature model in new way. This methodology allows class model to be declared in a way that reflects features model concepts with consistent mixing with feature model. It enhances configuration generation process to be simpler, more coherent and complete.

CHAPTER ONE: INTRODUCTION

1.1 Preface

Designing product lines process has received potential attention recently. This is due to the need of decreasing software product line steps and increasing system reusability.

Software Product Line (SPL) is the process of developing products' components from pre-defined core assets rather develop each component individually (Jézéquel, 2012).

Software Product Line (SPL) approaches attempt to increase system's productivity by designing a set of products that have many commonalities and shared characteristics, which leads to increasing system's reusability. On the other hand, SPL aims to identify and manage the variations among the products (Marco and Sybren, 2007).

Product line commonalities and variabilities are composed together in the *Domain Space* model as feature models, these models form the basic structure for future releases and system variant products (Jézéquel, 2012). A linked model named *Solution Space* is connected to the *Domain Space* to represent the real assets for variability elements associated with some rules to ensure valid selection and consistent system release generation (Marcilio et al., 2009). The relation between domain space and solution space is bi-directional; there is always a domain space needs a solution space, and for any solution, there is always a need to return back to the domain for better understanding. Figure 1-1 shows software product line spaces.

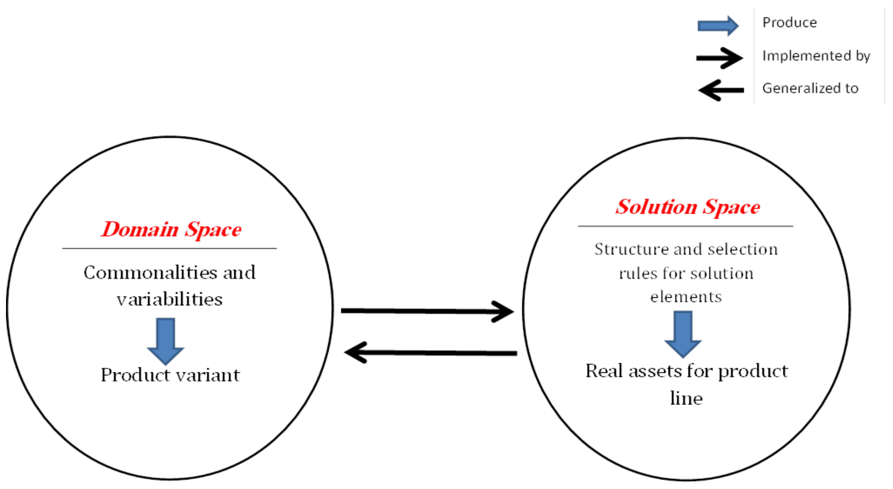

Produce
Implemented by
Generalized to
Domain Space
Commonalities and variabilities
Product variant
Solution Space
Structure and selection rules for solution elements
Real assets for product line

Figure 1-1 Software product line spaces

Several techniques are used to model domain space and solution space. Feature modelling is the most famous technique for this purpose (Jézéquel, 2012; Marco and Sybren, 2007). For modelling solution space, class models are used with some other options like Domain Specific languages (DSL) compilers, generative programs and configuration files (Laguna and Marques, 2009).

In the following sections, we present the context of our research, the problem for which we propose a solution, and motivation and contribution of this solution.

1.2 Research Context

This thesis deals with mixing classes and features modelling, so its research context shows SPL and variability approaches, object oriented (OO) approaches, and mixing class and feature modelling approaches.

Software product line and variability approaches*:* Over the past few years, several research contributions were reported to handle SPL variability process. They can be classified according to SPL's development methodology (*requirements, analysis, design, and implementation*) or the techniques they used to represent variability (*text, graph, or mixed*).

Approaches that support design and implementation steps (Gunther and Sunkle, 2012; Kacper, 2010; Kacper et al., 2011; Savinov, 2012; Stephan and Antkiewicz, 2008; Thaum et al., 2012) were developed to cover feature models that show the design phase of the product and class models that show the implementation phase.

Other approaches support SPL engineering in other steps like requirements and analysis. Alone, or in conjunction with others steps, these approaches (Acher et al., 2013; Asikainen et al., 2006; Gunther and Sunkle, 2012; Jézéquel, 2012; Marco and Sybren, 2007; Teixeira et al., 2011) presented variability by analysing the domain of the product and by the separation of concerns.

In order to handle these contributions, several techniques were developed. Techniques using graphical syntax and semantics were reported in (Jézéquel, 2012; Laguna and Marques, 2009; Sarinho and Apolinario, 2010; Sarinho et al., 2012; Stephan and

Antkiewicz, 2008; Teixeira et al., 2011). Others approaches that used text notations to represent variability were reported in (Classen et al., 2010; Ghoul, 2011). Finally, some researchers proposed mixed approaches (*graph notation and text notations*) like (Gunther and Sunkle, 2012; Kacper, 2010; Kacper et al., 2011).

Object-oriented modelling approaches: Approaches that used object-oriented paradigm (Savinov, 2012; Sim-Hui, 2013) to model variability described system architecture by package diagrams that used class diagrams. In order to understand these approaches, their main concepts are briefly introduced in this section.

Concept of class: A class is a set of specifications for a system's component (Sim-Hui, 2013). It defined the characteristics that this component may have, and the functionalities it provides. Over the past years, many approaches developed class models and object models and the relations between them to solve a lot of software programming domain problems.

Is-a hierarchy: one of the main concepts of object-oriented approach is the "is-a relation". It defines a child component as a "is-a" other component. Several problems were detected using this concept and reported in (Savinov, 2012; Sim-Hui, 2013). One of its main problem was the unnatural feature definition of child characteristics as parent characteristics.

Composed-by hierarchy: this approach was presented as a solution for the 'is-a" problems. It defines a component by composing other sub-components with different characteristics and methods. No inheritance relation between these components is defined using this approach. It is more natural and solves a lot of "is-a" approach problems.

Object (instances): Authors in (Savinov, 2012; Sim-Hui, 2013) defined object as a set of values for classes components. It is passed by a copy of class structure with final values added to it.

Mixing classes and features modeling approaches: Several approaches (Ghoul, 2011; Gunther and Sunkle, 2012; Kacper et al., 2011; Sarinho and Apolinario, 2010; Stephan and Antkiewicz, 2008) mix feature models with class models to present software product line engineering process. These approaches designed the variability and commonalities between variants of a product based on features with feature model, and implement these variations in class model. The mixing was done using several techniques like constraints additions (Gunther and Sunkle, 2012; Kacper, 2010), relation definition (Ghoul, 2011; Sarinho and Apolinario, 2010) and references links (Stephan and Antkiewicz, 2008).

These approaches defined the way for instantiating objects (configuration) that provides the final product (release) from selecting objects based on selected features and resolving constraints and relations among them.

Approaches supporting SPL requirement and analysis are good for providing general view of systems' needs and characteristics, but, they do not support system functionalities or structural behavioural like approaches covering design and implementation steps.

Graphical object-oriented modelling approaches provide clear representation for system hierarchy and components relations. While textual object-oriented approaches gives very strong semantic representation for system components and relations, but it is weak to represent the hierarchy relations and structure. Both textual and graphical object –oriented approaches are limited in modelling variability, because of absence of features.

Approaches that mix feature and class models encounter insufficient mixing techniques. These techniques do not provide powerful languages that mix system's feature and variability implementation (Jézéquel, 2012).

.

1.3 Problem Statement

From the above research context, the following challenges may be largely derived:

- Design and implementation approaches are very challenging phases, because they bridge between conceptual and implementation levels. Researches growth increasingly in this context.
- Variability design and implementation methodology which are poor if not absent. Their introduction and specification will lead to a great enhancement of SPL.

- Textual notation – based approaches are more formal syntactically and semantically than graphical approaches and more uniform than mixing ones.
- Mixing class and features approaches through new weak languages which are so far to be mature, evaluated, and accepted. Conceptual enhancements and practice evaluation will promote these valuables approaches to industrial level.
- Configuration generating approaches are complex and aiming to generate coherent and complete objects. Ensuring the simplicity, coherence, and completeness of these kinds of objects remain always as open problems.

1.4 Motivation

The work introduced in this thesis is stimulated by the following motivations:

- Lack of methodology supporting design and implementation of variabilities.
- Tackling the above challenges will allow SPL reaching high quality with moderate cost.
- Feature modelling has to be enhanced by adding meta-feature model classifying the features into main categories to reduce feature declaration and relation implementation.
- Class model should be specified in a way that reflects the feature model concepts and preserves its relations and constrains.
- Mixing feature model with class model has to be enhanced to guarantee fit representation of feature model and meta-feature model in class implementations.
- Configuration generation process has to be enhanced ensuring smooth and smart selection technique that respects feature’s rules and maintains old configuration for reuse.

1.5 Contributions

This thesis, propose new Textual Software Product Lines Design Model, mixing class and feature concepts, and aiming to bring significant solution elements to the previous problems, through its specific methodology:

- Provide a formal methodology supporting variability design and implementation. It bridges between product lines design model and object oriented implementation model.
- Provide a new concise and rich textual notation for feature modelling and class modelling.
- Allow simple and natural new way of mixing feature models and class models using small number of concepts and having uniform semantics.
- Allow simple, coherent, and complete configuration generation as simple class instantiation.

1.6 Thesis layout

In the following, we will start by presenting a case study which will be used through the entire thesis chapters, the literature review will be then introduced in chapter three. It will be oriented to identify insufficiencies that motivated our present work.

Our approach (A Textual Software Product Lines Design Model Mixing Class and Feature Concepts) will be presented in chapter 4, through the new developed methodology supporting variability design and implementation. This approach will be evaluated and compared with others' works in Chapter 5 in addition to a conclusion and expected future works.

CHAPTER TWO: CASE STUDY

2.1 Introduction

In this chapter, we will introduce a case study which will be used as a support to all our work. Our case study is to illustrate the idea of our approach and not to compute its value.

Class method's multiple definitions were introduced in several approaches like software design and subjective programming (Ghoul, 2011). In our case study, we will take "Set" product as an example.

Set product has several methods and attributes like Size, Data structure, *Empty()*, *Full()*, *Print()*, and Add(). Each of these can be implemented statically or dynamically. For an object of this product, it could use the static version of any method or the dynamic version. Thus, each method should be defined in two different ways; static definition and dynamic definition as shown in Figure 2-1.

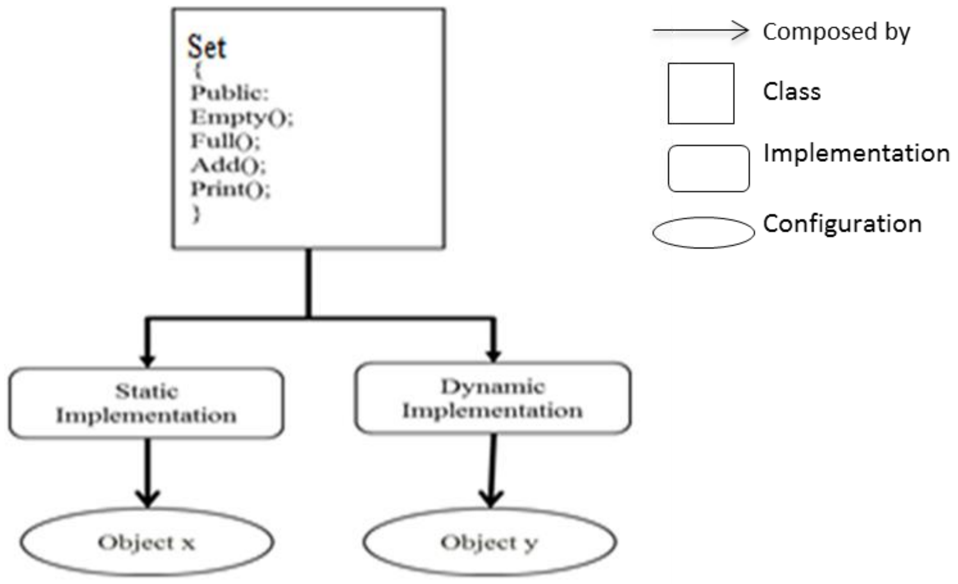


Figure 2-1 Multiple implementations of methods

Set product can be presented with two forms; Stack component and Queue component. Each of them has set's characteristics and its own characteristics. Each of these characteristics may be implemented statically or dynamically (Figure 2-2).

Implementing all these attributes and methods needs to be controlled, and the relations between them should be reserved during the implementations. The configuration process that requires selecting components with their implementation to create final reliable releases (such as stack and queue) should reserve the control relations too.

Some of these methods and attributes are shared for all releases configurations, like the Data structure and empty() method. Thus, their implementations should be in all releases which lead to multi-implementation.

2.2 Set's Features

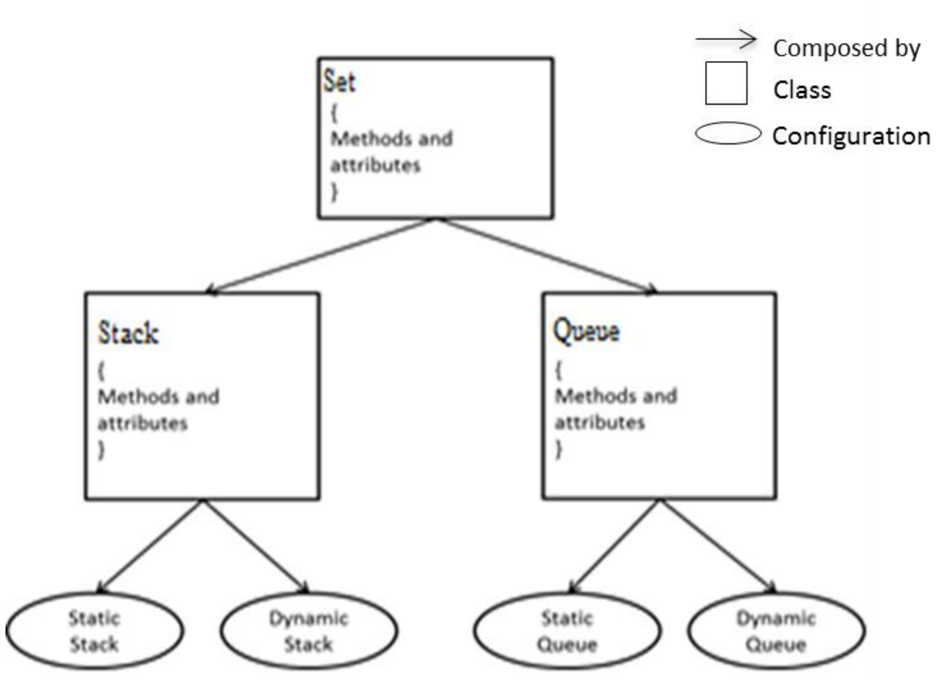

Composed by
Class
Configuration
Set { Methods and attributes }
Stack { Methods and attributes }
Queue { Methods and attributes }
Static Stack
Dynamic Stack
Static Queue
Dynamic Queue

Figure 2-2 Multiple definition with multiple sub-classes

To solve the problem of multi-implementation for methods and attributes, and to increase software maintainability and problem finding cost, feature implementation (Acher et al., 2013; Apel et al., 2013; Don, 2005; Kacper et al., 2011; Laguna and Marques, 2009; Thaum et al., 2012) was reported.

These features came from domain analysis, stakeholders' needs and many other sides that affect the implementation hierarchy. Some of these features affect other components. Some of them create new relations. And some of them shares specific characteristics that are applicable for all components of the system.

Set's features that can be extracted from its domain are View, Data structure, Scope, Behaviour, Order… etc. some of these features are shared everywhere in all releases that may be configured from set's components. For example, the View feature should be linked list or closed list in all releases.

Other features control the relations over set's components. For example, if the behaviour feature was static, this implies the data structure to be static.

If we will implement all set components and relations according to the feature they cover, the system will grow hugely, like shown in Figure 2-3.

Thus, we need to classify the features that the set component covers to reflect the global (shared) features, control features and other features that are included in configuration process.

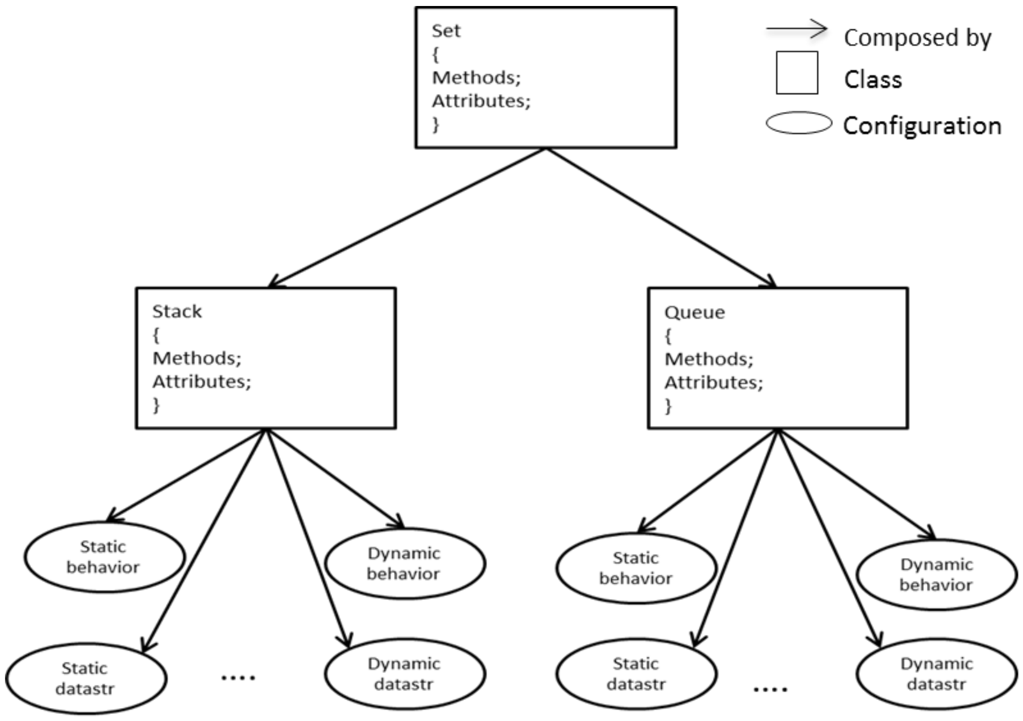


Figure 2-3 Features' implementations grow hugely

CHAPTER THREE: APPROACHES MIXING CLASSES AND FEATURES MODELS

3.1 Introduction

Large systems that are composed by huge number of different components cover multiple ideas and variant areas of interests. Thus, each of its components may have more than one possible value to cover. These values came from domain analysis, stockholders' needs, system evolution and so many other cases. The ability of a system to be generalized, specialized or customized (Marco and Sybren, 2007) to perform special needs is called system variability and specified using feature modelling.

In this chapter, we will review previous work that mixes class models and feature models for system variability. We start this chapter with listing feature modelling fundamentals and then overview approaches that mixes class and feature models.

3.2 Features modelling fundamentals

Over years of variability modelling, feature modelling using features diagrams was the most popular technique to represent variability in clear and meaningful way (Jézéquel, 2012)

Researches adopting feature modelling can be classified in three main groups based on the technique they used to present their feature models. These techniques are:

- *Graph notations based approaches*: Some approaches used pure graphical representation for their feature model's syntax and semantics like ECORE *(Stephan and Antkiewicz, 2008)* and OOFM (Sarinho et al., 2012), and the work reported by Laguna and Marques (2009), Razieh et al (2012), and Teixeira et al (2011).
- *Text notations based approaches:* Other approaches choose to use textual representation for their feature model's syntax and semantics like TVL (Classen et al., 2011) and FEATUREIDE (Thaum et al., 2012), and the work reported by Arnaud et al (2011).
- *Mixing text and graph notations:* In order to benefit from graphical and textual techniques, some approaches mixed them for representing their feature model.

These approaches like CLAFER (Kacper, 2010; Kacper et al., 2011) and RBFEATURES (Gunther and Sunkle, 2012).

In the following we will describe each technique and its main concepts.

Graphical feature modelling consists of tree hierarchy that shows the variable feature as the head node and the variant features as children nodes (Sarinho and Apolinario, 2010). The relations between these features mainly are:

- *Mandatory*: all children must be included in any configurations.
- *Optional*: this feature can be missed in the configuration.
- *Alternative* (Xor): exactly one of the children features is accepted.
- *Or*: at least one of the children features is accepted.
- *Propositional constrains*: specifies the dependencies relations between components.

Figure 3-1 shows these main relations graphically.

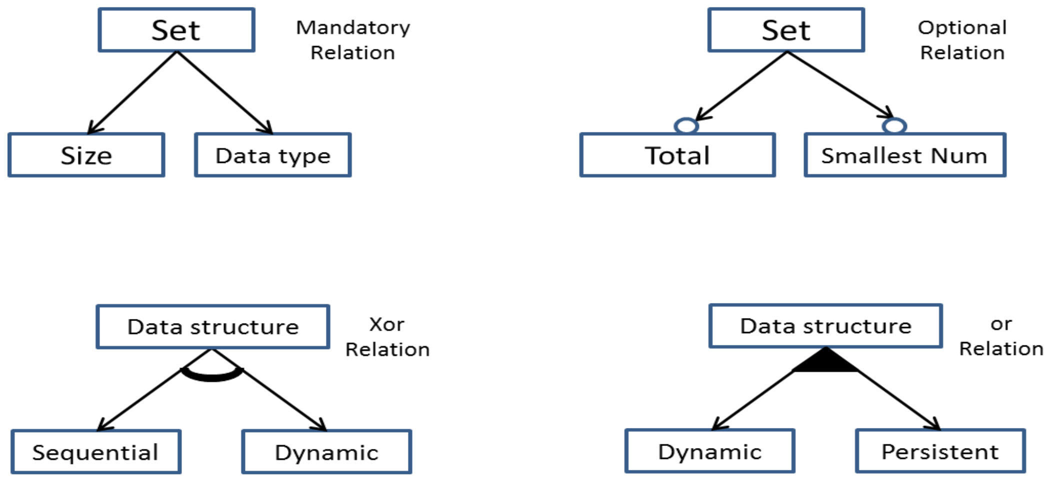


Figure 3-1 Feature modelling main relations

Graphical representation for feature models main concepts (Razieh et al., 2012) are:

- *Meta-Features Model*: Previous researches did not mention the Meta-Models Clearly. They mentioned it as features that may contain more than one sub features. We were the first to define Meta-Features Model as a design pattern that specifies feature's structure. It is applicable for all features and general for all kinds. Figure 3-2 shows a graphical representation for Meta-Feature Model.

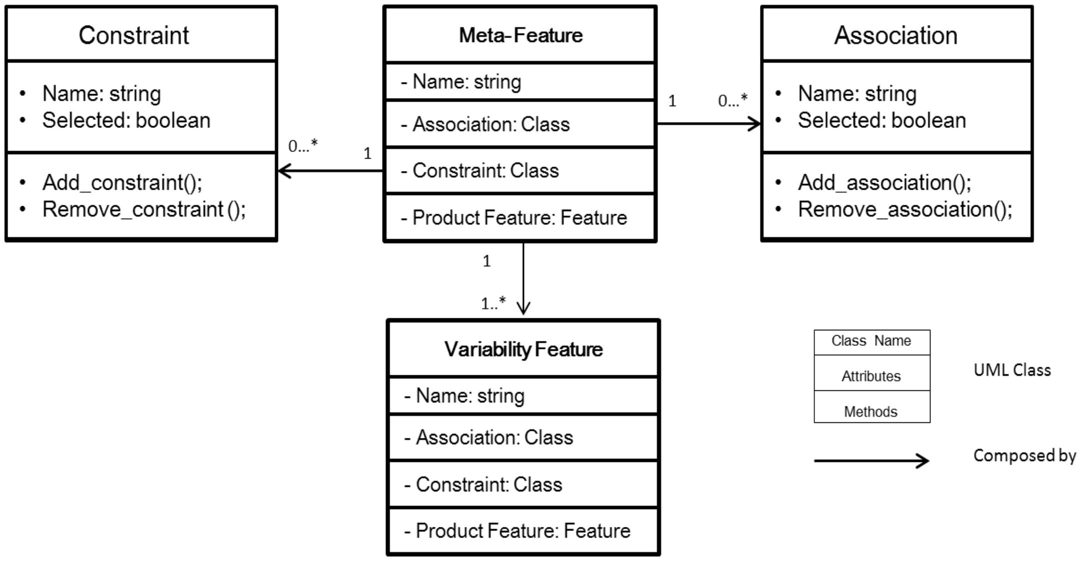


Figure 3-2 Meta-Features Model

- *Features Meta-Model*: Previous researches did not classify their features into categories that capture the main concepts in the feature model. We defined Features Meta-Model as a group that contains the main features that will be included in systems' release, and classified this model into four main categories. This model is predefined and domain independent (Figure 3-3).

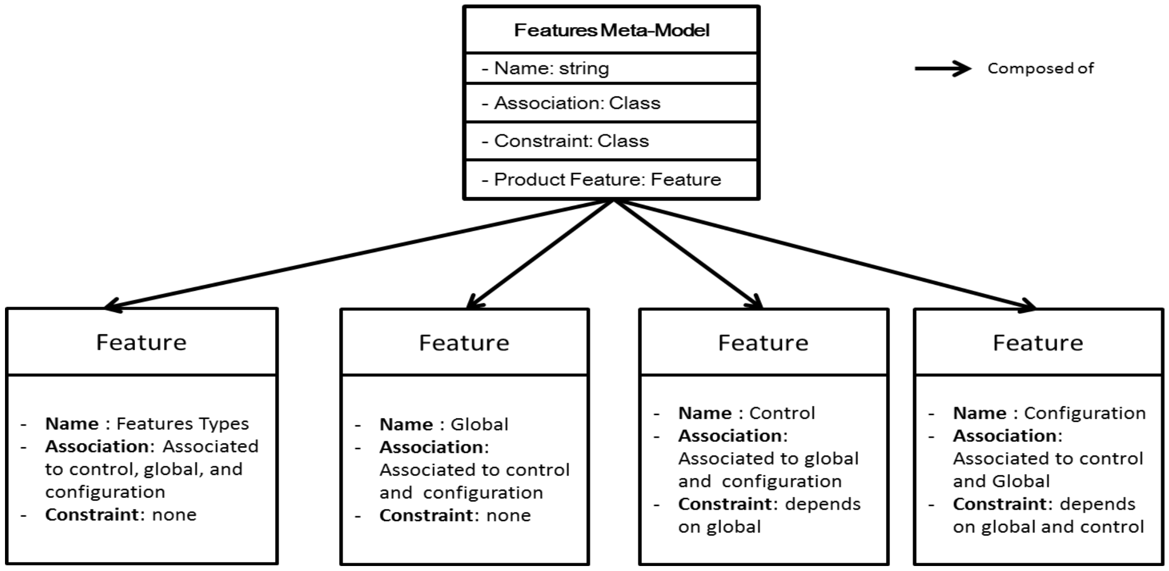

Features Meta-Model
- Name: string
- Association: Class
- Constraint: Class
- Product Feature: Feature
Composed of
Feature
- Name : Features Types
- Association: Associated to control, global, and configuration
- Constraint: none
Feature
- Name : Global
- Association: Associated to control and configuration
- Constraint: none
Feature
- Name : Control
- Association: Associated to global and configuration
- Constraint: depends on global
Feature
- Name : Configuration
- Association: Associated to control and Global
- Constraint: depends on global and control

Figure 3-3 Features Meta-Model

- *Feature Model*: Compact model of features diagram and feature constrains. It is an instance of the Features with Meta Model (Figure 3-4).

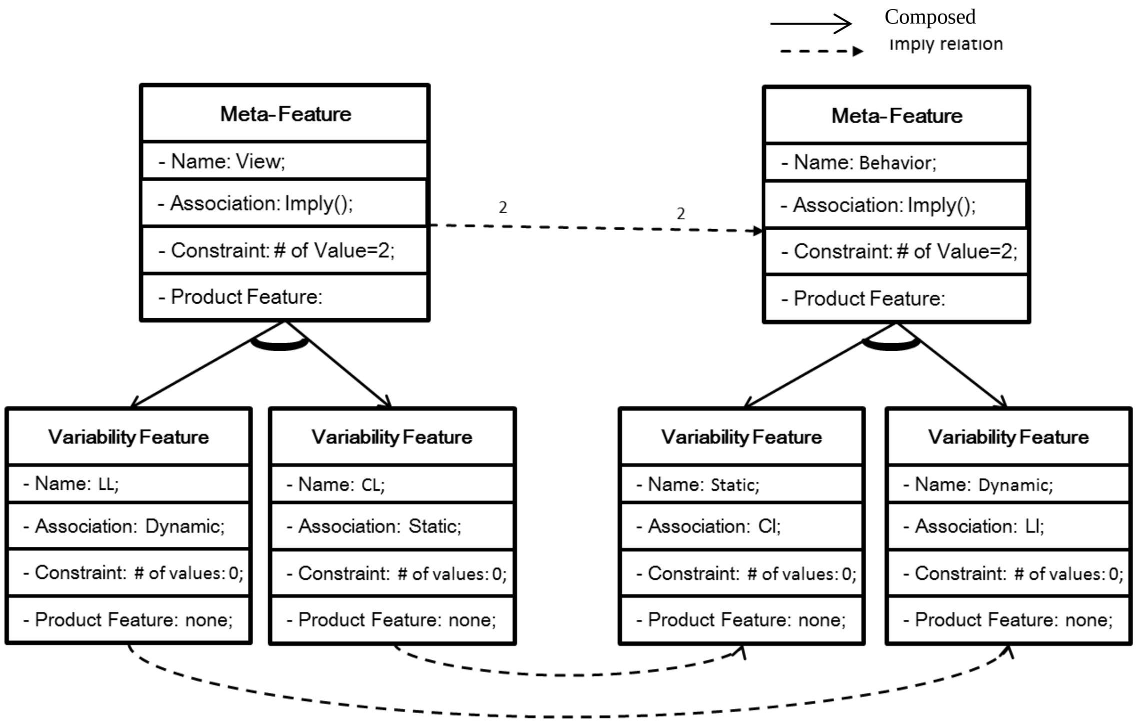


Figure 3-4 Feature Model

- *Feature diagram*: Graphical representation showing each feature and its relations with its subs.

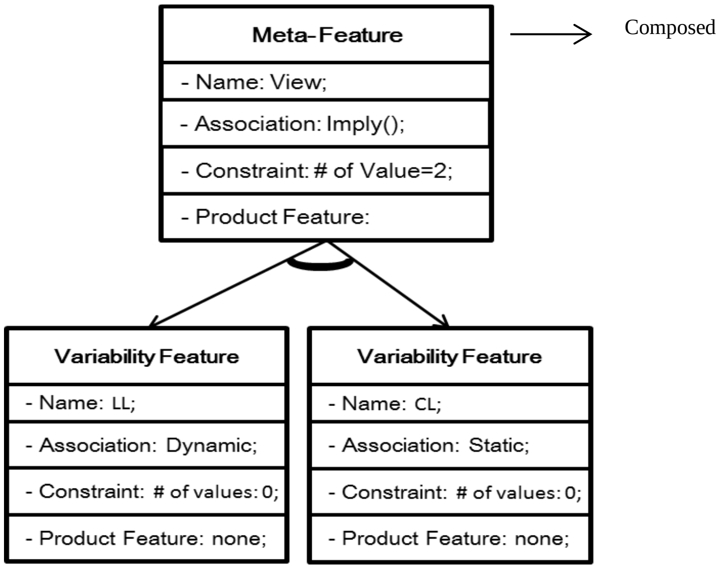
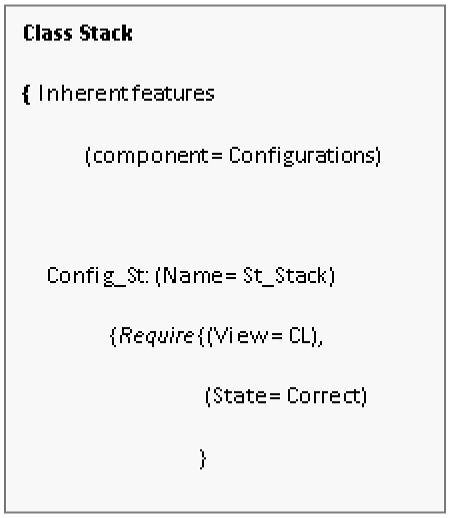


Figure 3-5 Feature Diagram

- *Feature's configuration*: Set of selected features producing a release in SPL. Configuration is permitted with feature model and preserves features' constrains. Figure 3-6 shows an example from our case study for features configuration.



```
Class Stack
{ Inherent features
        (component= Configurations)


   Config_St: (Name= St_Stack)
           {Require ((View= CL),
                        (State= Correct)
                       }
```

Figure 3-6 Features Configuration (stack example)

Designers do not prefer to use graphical representation for more than one reason (Classen et al., 2011): firstly, designing feature models using graphical representation is considered a very boring process and does not reflect the real semantic of system components. Secondly, graphical representation is very weak in representing system reasoning process (Marco and Sybren, 2007). Finally, graphical notation is still a "research prototype" (Classen et al., 2011) and can't reach text notations for representing feature models.

Textual feature model got rid of all these notations and modeling languages for representing features and their relations. They used simple texts composed by grammars, and propositional formulas (Arnaud et al., 2011) to show model structure and implementation.

Feature model's textual syntax was reported in several techniques like GUIDSL (Don, 2005) that represents feature models as grammars. This approach used by the AHEAD approach (Don, 2005) and FeatureIDE approach (Thaum et al., 2012).

Other techniques like SXFM file format (Marcilio et al., 2009), XML and The VSL file format of the CVM framework (Reiser, 2009) were used to represent the meta-models, and supported by textual feature models .

Some approaches prefer to mix graph notations with text notations to achieve the best benefits from both of them.

CLAFER in (Kacper, 2010) presented their feature model as graph notations and presented a textual representation for their class model. this was the same case for work presented in (Kacper et al., 2011) and RBFEATURES approach presented in (Gunther and Sunkle, 2012).

3.3 Models mixing classes and features

Feature modelling used to design system's variability and communality over its components (Kacper, 2010). Class models capture the implementation part of the products by showing the real values and relations over components' attributes. Thus, mixing both models (feature model and class model) provides the full picture for SPL's components.

In this section, we will review the literature works mixing feature models with class models in two phases:

- *How they mix feature models and class models?*
- *How they instantiate objects (configuration) to create final products***?**

CLAFER model (Kacper, 2010) presents a good approach for mixing class model with feature model based on constraints and inheritance concepts. The feature model was presented as a collection of type definitions and features (Figure 3-7).

```
Abstract Set
    Size : integer

Abstract stack extends Set
Abstract S_stack extends Set
     Behavior: static
      [Size>=1]
```

Figure 3-7 CLAFER Example for Mixing Class and Feature

The mixing between feature model and class model via constraints is added to class model as attributes and attributes' values. The final model is restricted to one configuration based on the mixed feature. Object instantiation in CLAFER is done by adding constraints to the feature model resulting as constrained feature model. These constraints restrict the feature model to single or dual configuration presenting one or more final product (Figure 3-8).

```
Concret product
Datastr==Dynamic && View==LL;
```

Figure 3-8 CLAFER Configuration Instantiation

Gunther and Sunkle (2012) reported feature oriented programming language called RBFEATURES on top of dynamic programming language (ruby).

The class model was reported as a first-class entity and named *ProductLine*. Mixing feature model with class model was done via add-feature method.

After creating feature model in RBFEATURES, the *ProductLine* that is created via *configure* method and collects number of conceptual features. It is allowed to set specific feature configuration with *activate_feature* and *deactivate_feature* operations. Final result is represented in the variable called *@feature_tree* which is used in a method called instantiate that creates object after checking some mandatory constraints that guarantee consistent final product.

Sarinho and Apolinario (2010) presented object-oriented feature model that combined feature models' concepts with object-oriented concepts. They proposed object-oriented feature model (OOFM) profile that is composed by feature model and feature modelling package.

Feature classes were reported in (Sarinho and Apolinario, 2010) with object-oriented relationships and resources to provide new level of variability documentation. Feature classes can be declared using *feature-class stereotype* that creates classes according to designer's intentions. This process composed by several steps starting by feature package creation, followed by OOFM profile mapping and ended by class feature declaration.

Bio-inspired aspect-oriented paradigm was presented by Ghoul (2011) to reflects biological principles on the artificial systems. The author presented aspect models as *Genomes* components and class models that implement them. The mixing was done using relation between feature models and class models. Object instantiation is done by a WEAVER component that guarantees the consistency over all components. After that an adapted design interface will be created and a given object name will be defined. Figure 3-9 shows an example for bio-inspired model.

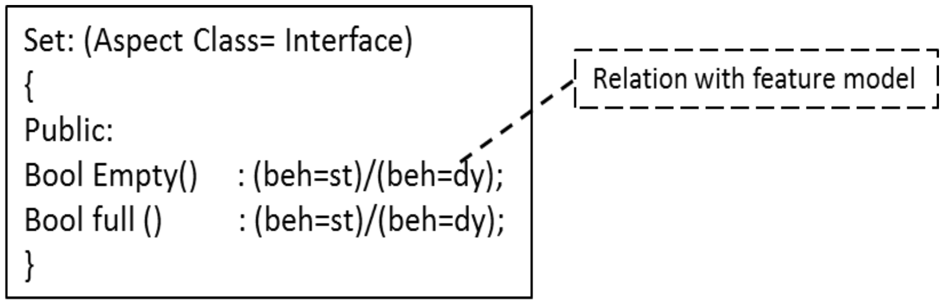

```
Set: (Aspect Class= Interface)
{
Public:
Bool Empty()     : (beh=st)/(beh=dy);
Bool full ()     : (beh=st)/(beh=dy);
}
```

Figure 3-9 Bio-inspired Class and Aspect Model

Stephan and Antkiewicz (2008) reported ECORE , a class model notations that are presented as feature models. It is composed of meta-model that is created from class model using ECORE itself. Class model is composed by several sub-classes that are composed by other sub-classes. Mapping between feature models and class models was done in both ways: feature to class mapping and class to feature mapping.

Class to feature mapping requires implementing all class model notations as feature notations. This is done by sequential steps mentioned and described in (Stephan and Antkiewicz, 2008). The opposite mapping is done by specialization steps for feature model to create class model based on designer intentions using commands like add, remove, and modify.

Object model provides a conceptual view of the final product to give designer basic structure of configuration model. Features in the configuration are presented as children to abstract features in feature model. The final set of configuration features is considered as a prototype for object model.

CLAFER (Kacper, 2010) did not mention multiple feature connections and the contrary relations that may arise during the mixing. Weak representation of features' possible values that may construct the feature model was found in CLAFER.

Feature classification was missed in (Gunther and Sunkle, 2012). And there was no mention for the relations between these features. The class was defined based on configuration only not based on the features. This makes the process of tracking features' objects hard.

The OOFM that was introduced in (Sarinho and Apolinario, 2010) and extended in (Sarinho et al., 2012) did not provide a separation between feature and object model. This leads to an entangled system.

Aspect-oriented approach (Ghoul, 2011) is restricted for aspect-oriented programming systems, and may not be applicable for all object-oriented programming systems.

Defining class model and extending it to feature model means that features are restricted to class model. Adding, removing or modifying features will be hard process since class model has to be modified each time. Thus, ECORE (Stephan and Antkiewicz, 2008) tool is not efficient in separating concerns for system features and implementation. Design and implementation methodologies in the previous approaches are weak or absent.

3.4 Thesis contribution

Based on weaknesses mentioned in the literature works presented in the previous section, we are proposing our model to enhance the actual state of the research domain.

In order to make product line engineering process more natural and simple, and to capture object oriented approaches' benefits, we propose a software engineering methodology bridging product lines design model and implementation model for creating object oriented SPL and specifying its introduced concepts; Meta-Feature Model, Feature Meta-Model, Feature Model, Product Meta-Model, Product Model.

After studying feature modelling techniques, we found that textual models have more advantages than graphical techniques. Thus, we provide a concise and rich textual notation for feature modelling and class modelling.

This feature model has to be linked with class model in a way that reflects features' concepts. Thus we will allow simple and natural mixing feature models and class models using small number of concepts and having uniform semantics.

Finally, we allow simple, coherent, and complete configuration generation as simple class instantiation.

CHAPTER FOUR: A TEXTUAL MODEL MIXING CLASSES AND FEATURES

4.1 A Textual Design Methodology (TDM)

In this section we present our approach for modelling features in SPL systems. We are aiming to increase system modularization by separating concerns from the variability components. Thus, four main meta-features were created.

We used the separated approach (Istoan, 2013; Jézéquel, 2012) to represent our model where the Product model is represented separately from the Feature Model.

In the following, we introduce our textual SPL design methodology (TDM), its features concepts, its object-oriented concepts, its mixed class and features concepts, its illustration by our case study, and finally a conclusion on its specification.

The TDM, with graph notations showing its ordered steps for designing variable software, is shown in Figure 4-1. Graph notations are used only for clarity purposes and not as syntactical.

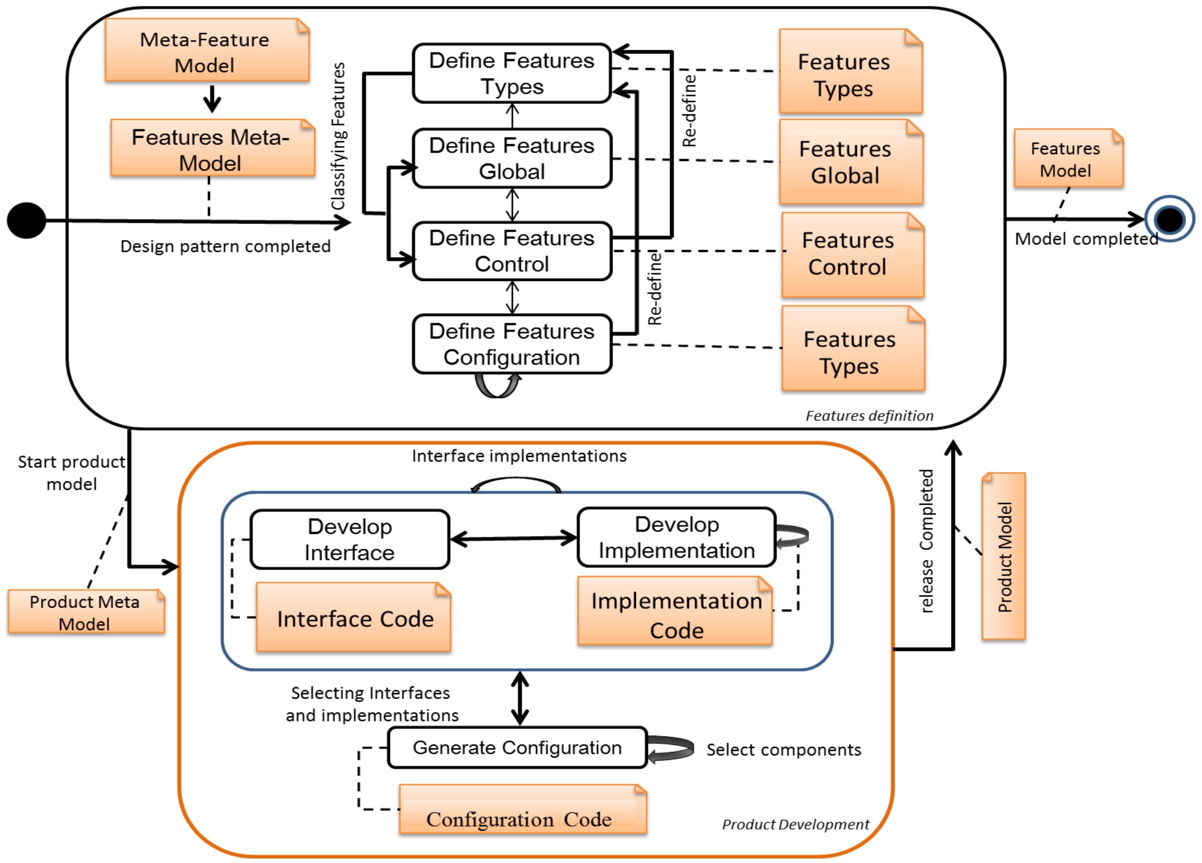


Figure 4-1 Textual Design Methodology (TDM) mixing class and feature concepts, using UML state diagram notations

4.2 TDM Features Concepts

In the following, we will present TDM steps. Designing steps are based on pre-defined features. A new development will be started by instantiating the *Features Meta- Model*. This model is composed by four features: *Features types*, *Features Global*, *Features Control and Features Configuration.*

1. *Meta-Features Model*: It is a predefined design pattern that defines all features in TDM. It is the base for features in *Features Meta-Model*. The graphical structure is shown in Figure 3-2 (repeated in Figure 4-2).

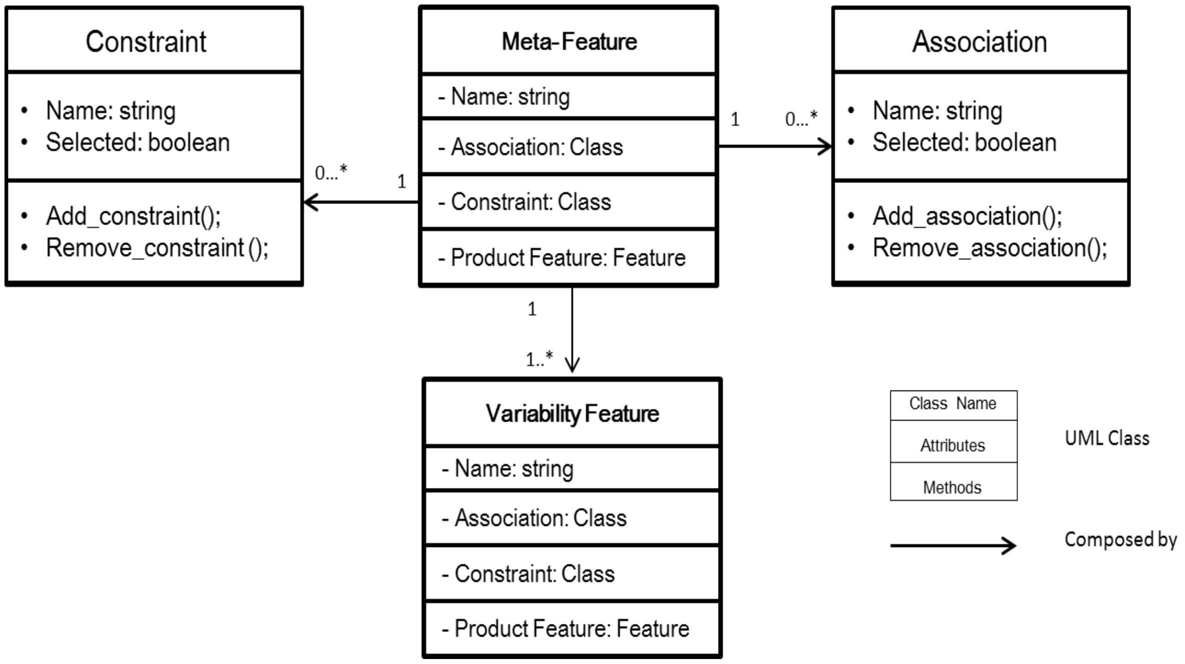


Figure 4-2 Graphical representation for Meta-Features Model

Figure 4-3 shows the textual representation for this model. Each feature is composed by a name; to distinguish it from other features, an association component to determine its associations with other features, a constrain component that specifies constraints may affect its relations with others, and finally, a Product features that form the real features for it.

```
Meta-Features Model
{
Name: String;
Association: Class;
Constraint: Class;
Product Feature: Feature;
}
```

Figure 4-3 Meta-Features Model (textual representation)

Features Meta-Model: It is the input features design pattern to the methodology. It is predefined based on *Meta-Features Model* design pattern. It is domain independent, and we instantiate feature model (which is domain dependent) from it. Figure 3-3 (repeated in Figure 4-4) shows a graphical representation for this model, while Figure 4-5 shows the textual model.

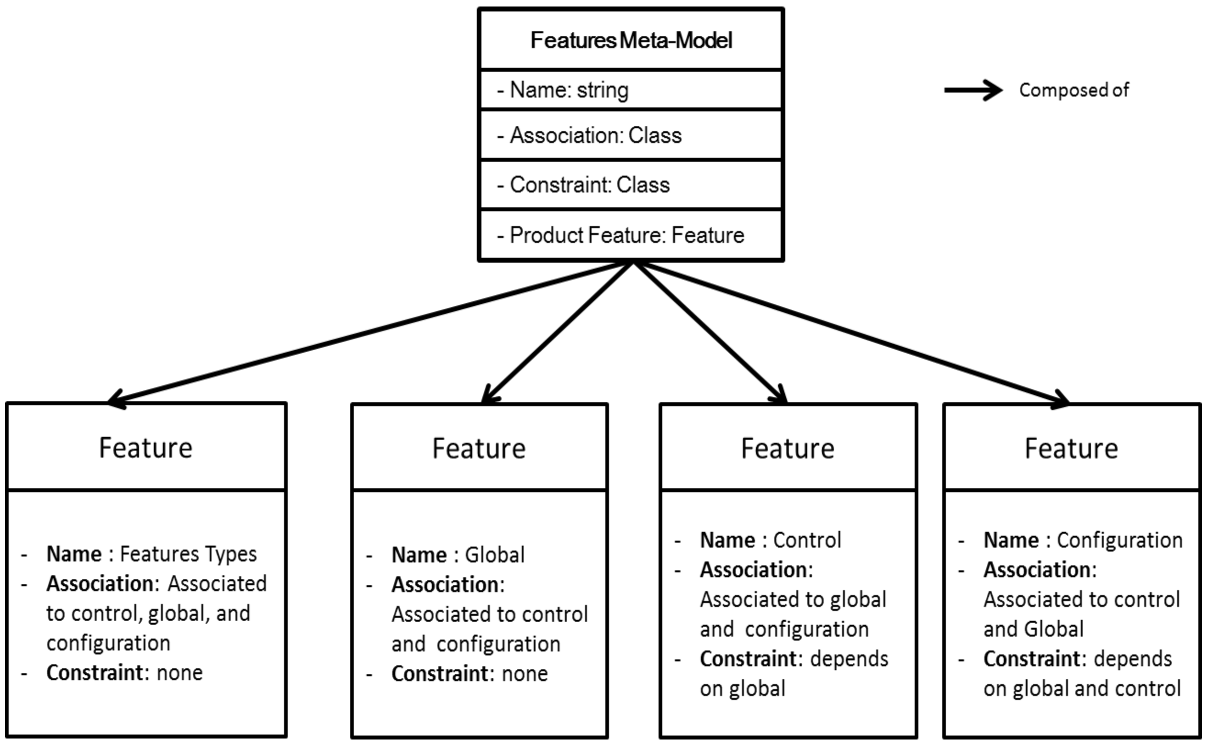

Features Meta-Model
- Name: string
- Association: Class
- Constraint: Class
- Product Feature: Feature
Composed of
Feature
- Name : Features Types
- Association: Associated to control, global, and configuration
- Constraint: none
Feature
- Name : Global
- Association: Associated to control and configuration
- Constraint: none
Feature
- Name : Control
- Association: Associated to global and configuration
- Constraint: depends on global
Feature
- Name : Configuration
- Association: Associated to control and Global
- Constraint: depends on global and control

Figure 4-4 Features Meta-Model (repeated)

```
Features Meta-Model
{
Name: string;
Associations: Class;
Constraint: Class;
Product Feature: Feature;
}
```

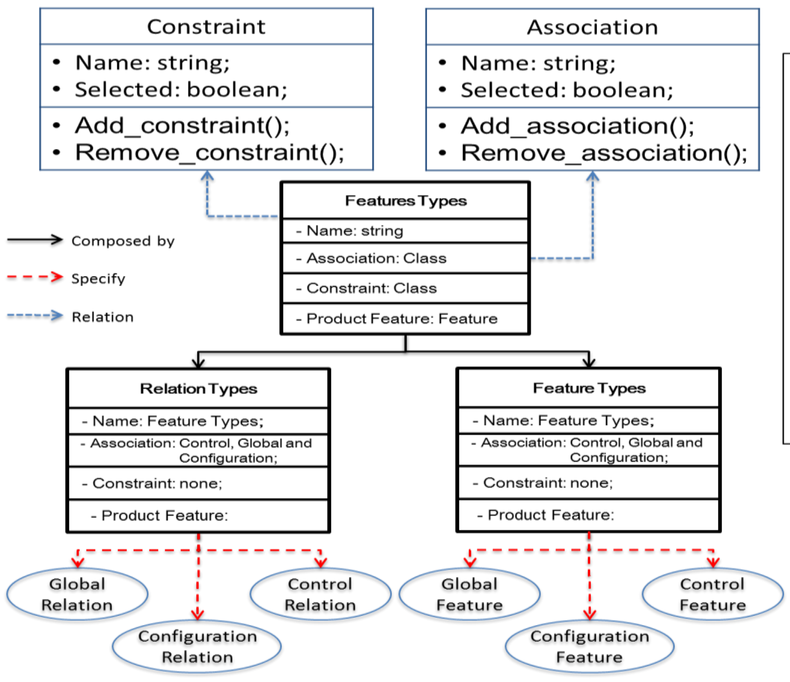
```
Product Feature
{
Features Types: Feature;
Features Global: Feature;
Features Control: Feature;
Features Configuration: Feature;
}
```

Figure 4-5 Features Meta-Model (textual representation)

Below, each feature is presented separately showing its graphical and textual representation.

2. *Features Types*: This composed (class) feature captures all features (relations and features) in the system with their concrete values. It is composed by *Features_ Types* and *Relation_ Types*. The former represents all systems' features (characteristics). And the later represents all systems' possible relations.

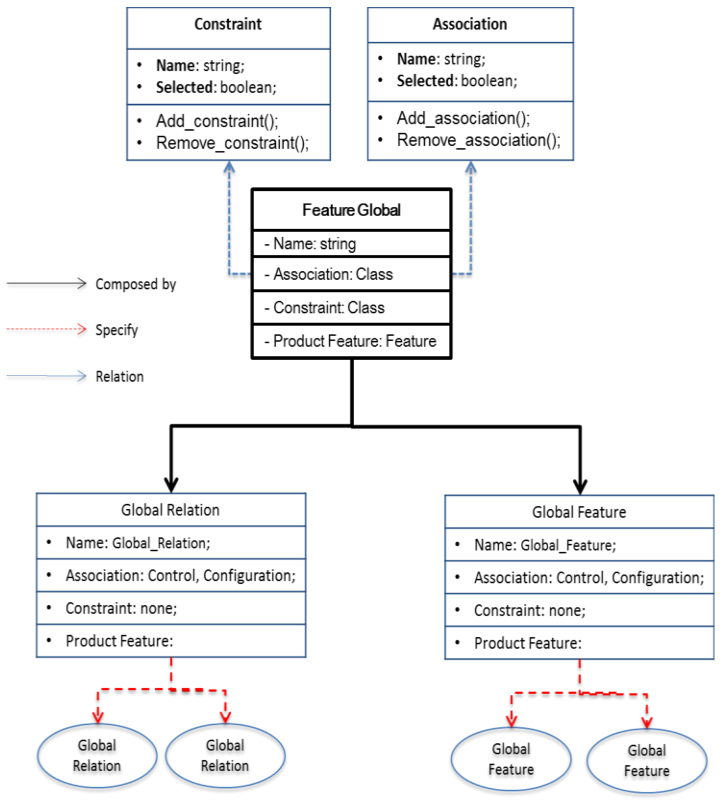
These features and relations will specify the *Global*, *Control and Configuration features*. Figure 4-6 shows graphical and textual representation of Features Types.



```
Feature Features Types
{
Name: Features Types;
Associations: Control, Global,
              Configuration;
Constraint: none;
Product Feature Feature_Type;
Product Feature Relation_Type;
}
```

Figure 4-6 Features Types

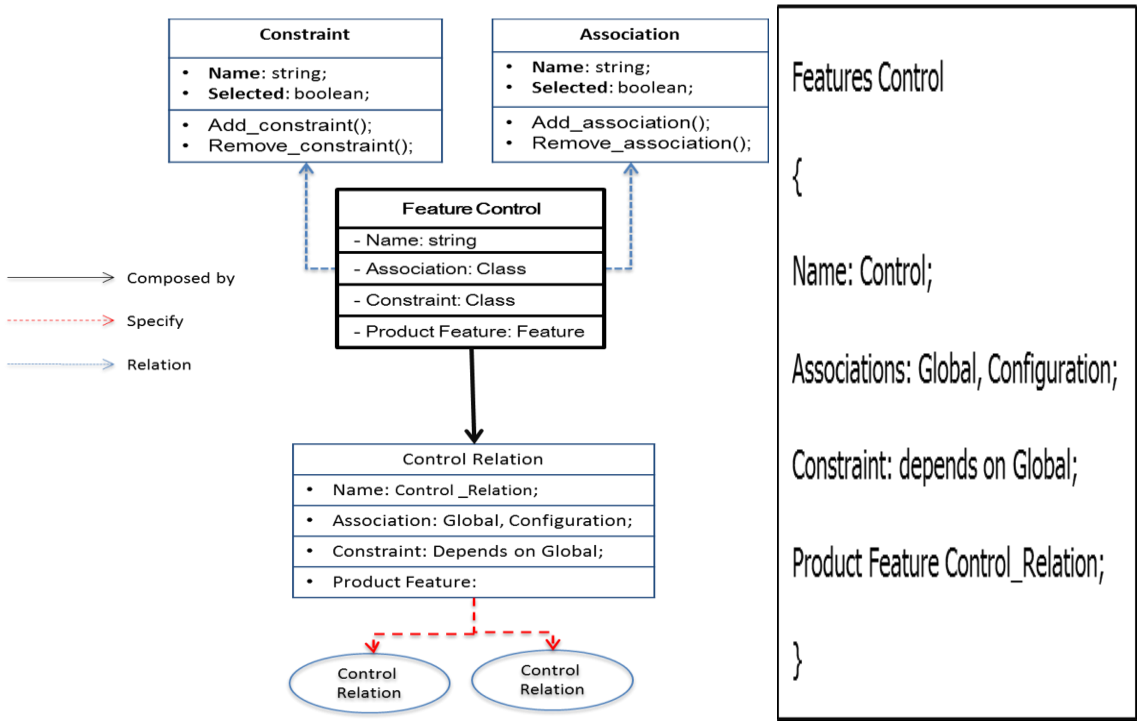
3. *Features Global*: This composed feature specifies the *Global* features that will be shared for all system components. *Global* features may be relations over components or just features (characteristics) that must be applicable everywhere. Figure 4-7 shows the textual and graphical representation for *feature Global*.



```
Features Global
{
Name: Global;
Associations: Control, Configuration;
Constraint: none;
Product Feature Global_Relation;
Product Feature Global_Relation;
}
```

Figure 4-7 Features Global

4. *Features Control:* This composed feature specifies the controls over all systems' components and relations. Any configuration should reserve control's relations to ensure system consistency. This feature is composed by relations only, and its main goal is to keep systems' components stable and avoid any conflicts. Figure 4-8 shows graphical and textual representation for *feature Control*.

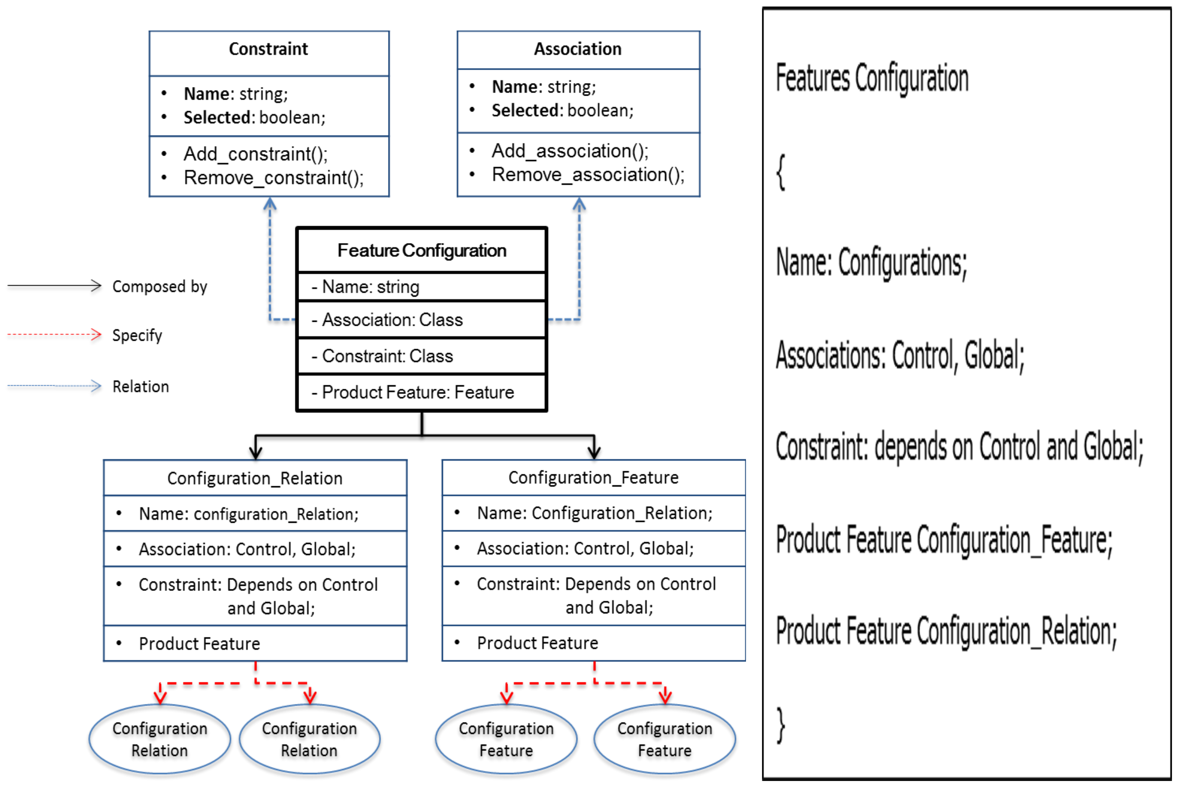


```
Features Control

{

Name: Control;

Associations: Global, Configuration;

Constraint: depends on Global;

Product Feature Control_Relation;

}
```

Figure 4-8 Features Control

5. *Features Configuration*: This composed feature specifies required and discarded features for a product configuration (release). Figure 4-9 shows the graphical and textual representation for Feature Configuration.

Features Types, Global, Control and Configuration together compose the Features Meta-Model of TDM. The second step is creating Feature Model.



```
Features Configuration

{

Name: Configurations;

Associations: Control, Global;

Constraint: depends on Control and Global;

Product Feature Configuration_Feature;

Product Feature Configuration_Relation;

}
```

Figure 4-9 Feature Configuration

6. *Features Model*: This is an intermediate model between the conceptual part (*Feature Meta-Model*) and the physical part (*Product Model*).

In this model, all features and relations in the Features Meta-Model are instantiated. These features and relations are software-dependent. A clear view will be provided in the fourth section.

Figure 4-10 shows instantiation of Features Model from Features Meta-Model. Meaning that each Features Meta-Model may have one or more instances in its Features Model. Thus, the cardinality relation between them is one to many.

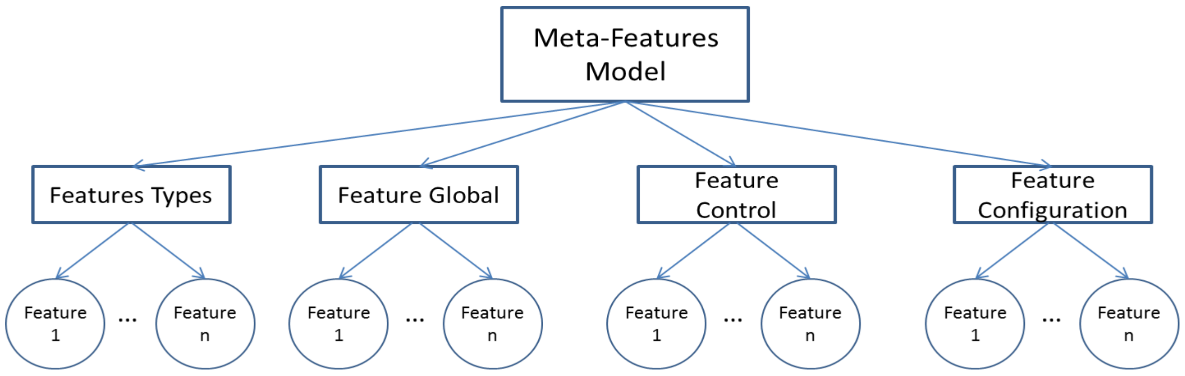


Figure 4-10 Instantiation of Feature Model from Feature Meta-Model

4.3 TDM Object-oriented Concepts

In this section, we will report the object-oriented concepts that TDM covers through its Product Meta-Mode (Figure 4-11,Figure 4-12).

Class Interface specifies services provided by a product component. It includes its provided methods, its attributes (data) and its different implementations' list. Figure 4-11shows the graphical representation for Class interface, and Figure 4-12 shows the textual representation.

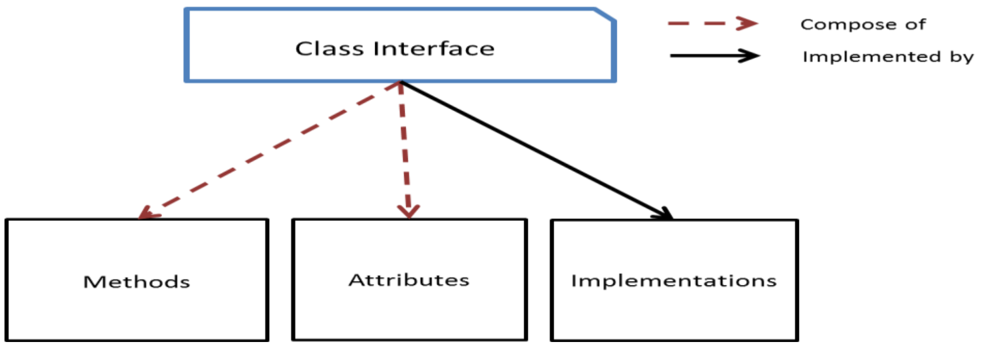


Figure 4-11 Graphical representation for class interface

```
Class Interface <name>
{
Attributes;
Methods;
Implementation;
}
```

```
Class Implementation <name>
{
Attributes implementation;

Methods implementation;
}
```

Figure 4-12 Class Interface and Class implementation (textual representation)

4.4 TDM Mixing Class and Features Concepts

This section exposes the mix class and features concepts that TDM covers through its *Product Meta-Model* and *Product Model*.

1- Product Meta-Model: It is the product meta-model of object-oriented paradigm mixed with features (defined from domain), and inherent features (that is defined for each component based on its properties). It is composed by Interface Meta-Model and Implementation Meta-Model as shown in Figure 4-13 and Figure 4-14.

Each attribute or method can be defined in several ways depending on the features it composes. Each time a new feature is added to an attribute, a new definition should be held.

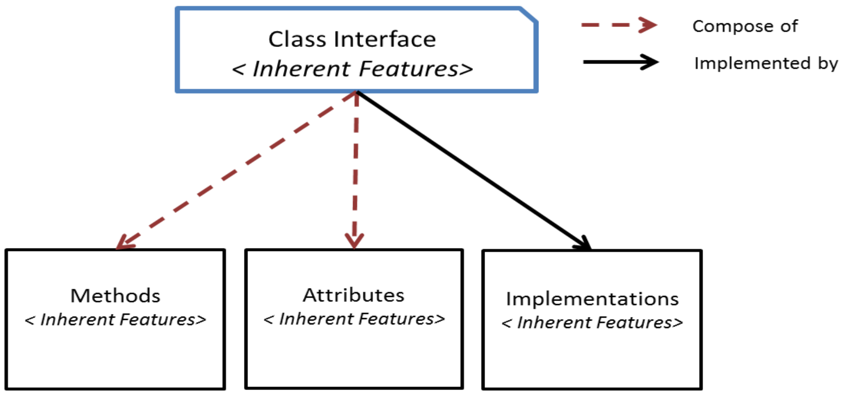


Figure 4-13 Interface Meta-Model (graphical representation)

```
Class Interface <name>
{
Inherent features     < ...>;
Attributes definition <...>;
Methods definition  <...>;
Implementation Link;

}
```

```
Class Implementation<name>
{
Inherent features     < ...>;

Attributes <...>;

Methods <...>;
}
```

Figure 4-14 Interface and Implementation Meta-Model (textual representation)

2- Product Model: This is the final model. It is composed by class interfaces and their implemented attributes, methods and implementations. Figure 4-15 shows the graphical representation for this model, and Figure 4-16 shows the textual representation.

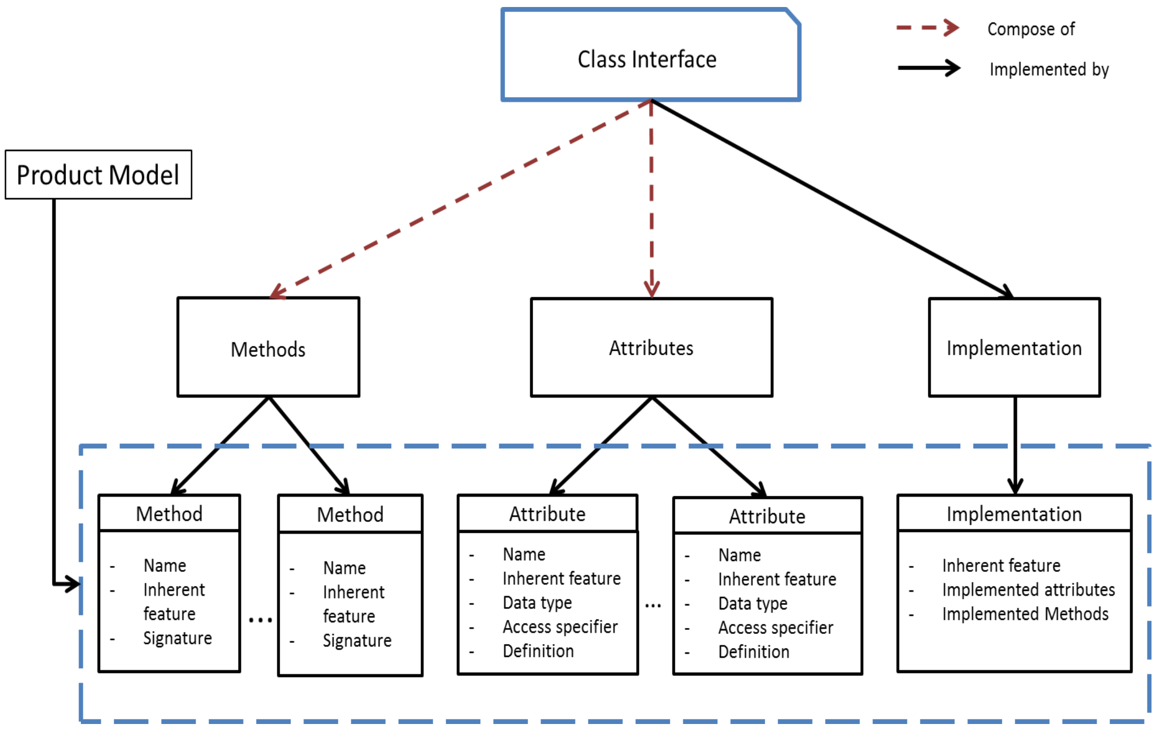


Figure 4-15 Product Model (graphical representation)

```
Product Model
{
Interfaces;
Implementations;
}
```

Figure 4-16 Product Model (textual representation)

The full model is presented in Figure 4-17, it shows the composed Feature Meta-Model, Feature Model and Product Model.

This model is inspired from TDM figure (Figure 4-1) with more details about the components and Meta- features. The first two parts from this model (Features Meta-Model and Features model) corresponds to the first state in TDM (feature definition), while the last part (Product Model) corresponds to the second state in TDM (Product development).

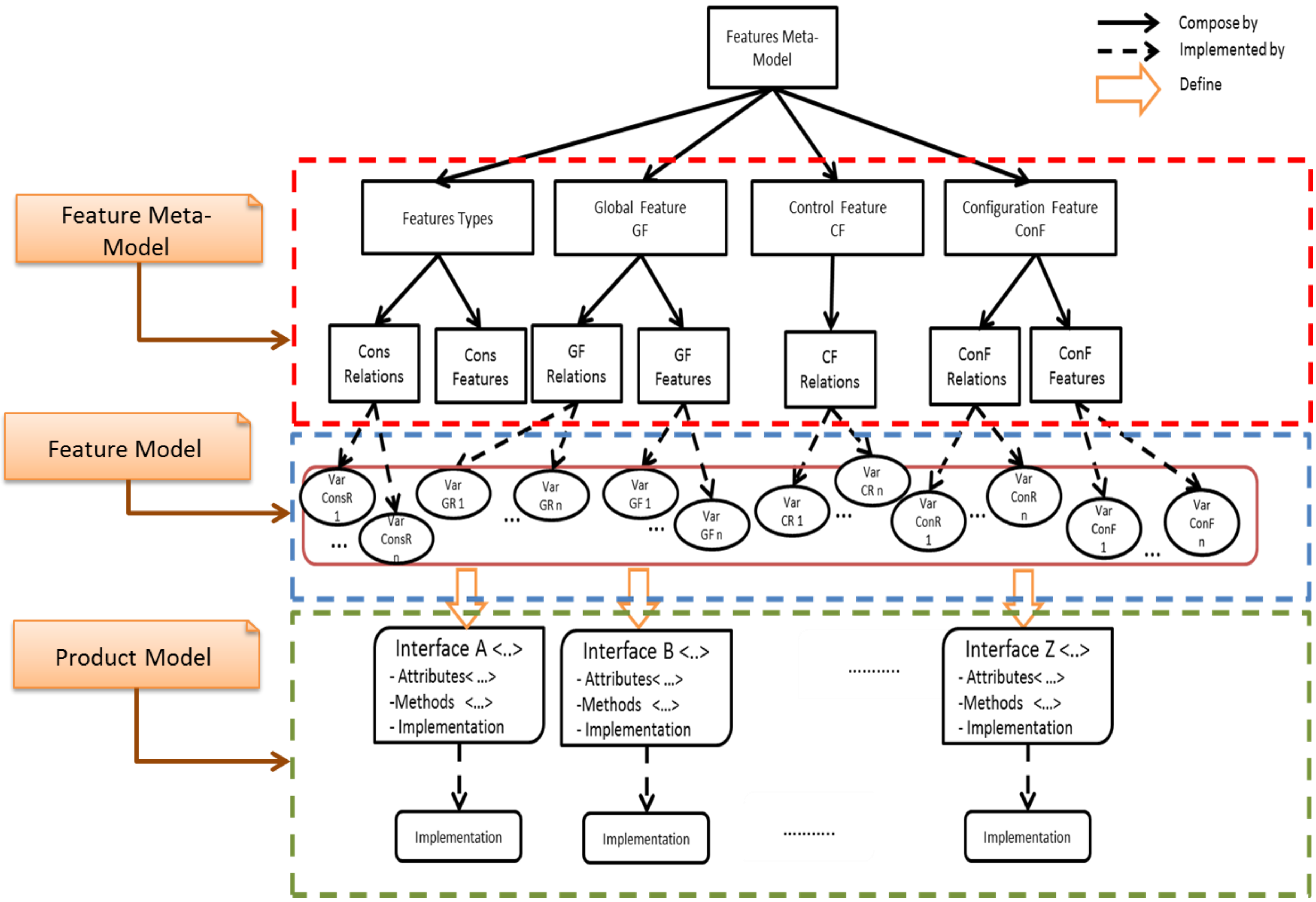


Figure 4-17 Full Model mixing feature and class concepts

4.5 A product Instance: case study

Our case study was reported in chapter 2 of this work.

Set model has several implementations such as: Static stack, static queue, dynamic stack and dynamic queue.

1- *Feature Model*: In the following, we present the Feature Model of the "Set Model", composed by its *Features Types, Feature Global, Feature control and Feature Configuration.*

The first Feature in "Set" Feature Model is *Features Types*. It defines all the features in the system with all their possible values.

Figure 4-18 shows the graphical representation for the Features Types, and Figure 4-19 shows the textual representation.

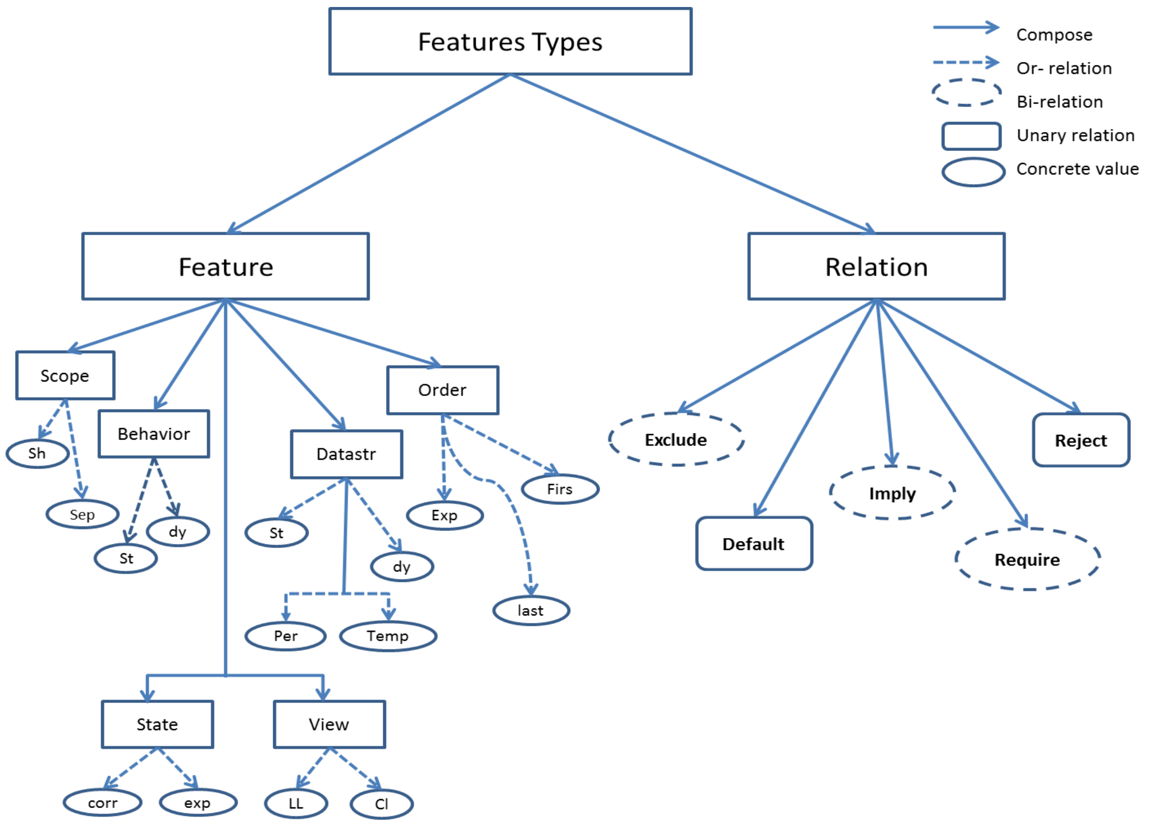


Figure 4-18 Set Feature Types (graphical representation)

```
Features Types:
{
Name: Relation_Type;
Type: FTR;
Product Relation Exclude;
{
Exclude.name=Exclude;
Exclude.Type=bi;
}
Product Relation Defualt;
{
Defualt.name= Defualt;
Default.type=Unary;
}
Product Relation Imply;
{
Imply.name= Imply;
Imply.Type=bi;
}
Product Relation Require;
{
Require.name= Require;
Require.Type=bi;
}
Product Relation Reject;
{
Reject.name= Reject;
Reject.type=Unary;
}
}
```

```
Features Types:
{
Name: Feature_Type;
Type: FTF;
Product Feature Scope;
{
Scope.name=Scope;
Scope.Num_of_values=2;
Scope.values[1]=shared;
Scope.values[2]=separated;
}//end of Feature_Type Scope

Product Feature Behavior;
{
Behavior.name= Behavior;
Behavior.Num_of_values=2;
Behavior.values[1]=Static;
Behavior.values[2]=Dynamic;
}//end of Feature_Type behavior

Product Feature State;
{
State.name= State;
State.Num_of_values=2;
State.values[1]=correct;
State.values[2]=experimental;
}//end of Feature_Type state
```

```
// Features Types:continue
Product Feature View;
{
View.name= View;
View.Num_of_values=2;
View.values[1]=Ll;
View.values[2]=Cl;
}//end of Feature_Type view

Product Feature Datastr;
{
Datastr.name= Datastr;
Datastr.Num_of_values=4;
Datastr.values[1]=static;
Datastr.values[2]=dynaic;
Datastr.values[3]=persistent;
Datastr.values[4]=temporary;
}//end of Feature_Type Datastr
```

```
// Features Types:continue
Product Feature Order;
{
Order.name= Order;
Order.Num_of_values=3;
Order.values[1]=experimental;
Order.values[2]=first;
Order.values[3]=last;
}//end of Feature_Type order

Product Feature Form;
{
Form.name= Form;
Form.Num_of_values=2;
Form.values[1]=ch;
Form.values[2]=con;
}//end of Feature_Type Form
}
```

Figure 4-19 Set Features Types (textual representation)

The second feature is *Features Global*. It captures model global characteristics and relations Figure 4-20 shows the graphical representation for global feature, and Figure 4-21 shows the textual representation.

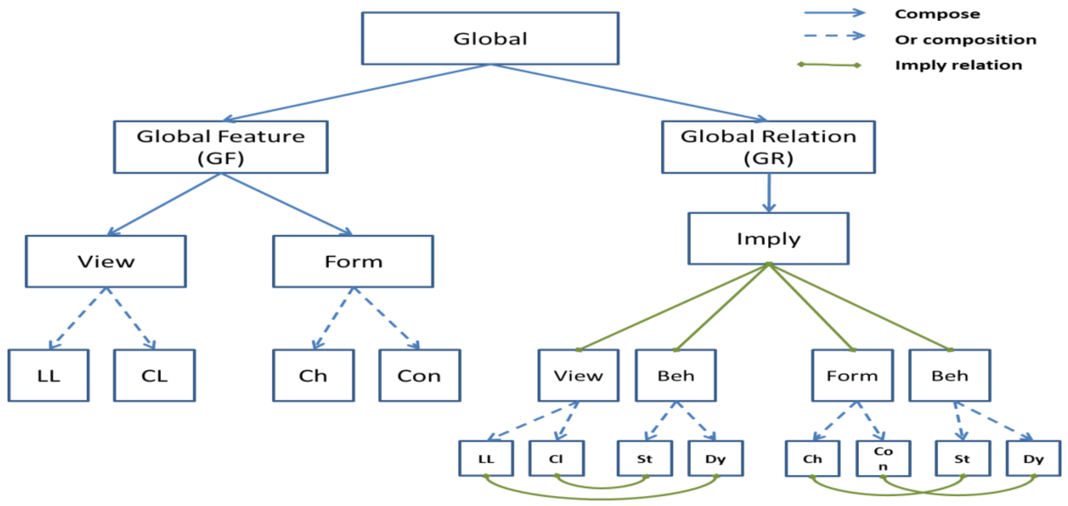


Figure 4-20 Set Global Feature (graphical representation)

```
Features  Global
Name: global_Relation;
Type: GR;
Product Relation Imply_1;
{
Imply_1.parts[1]=view.Ll;
Imply_1.parts[2]=Behavior.dynamic;
}
Product Relation Imply_2;
{
Imply_2.parts[1]=view.Cl;
Imply_2.parts[2]=Behavior.static;
}
Product Relation Imply_3;
{
Imply_3.parts[1]=Form.ch;
Imply_3.parts[2]=Behavior.static;
}
Product Relation Imply_4;
{
Imply_4.parts[1]=Form.con;
Imply_4.parts[2]=Behavior.dynamic;
}
```

```
Features  Global
{
Name: global_feature;
Type: GF;
Product Feature   view;
Product Feature   Form;
}
```

Figure 4-21 Set Global Feature (textual representation)

The third feature is *Featurse Control*. It is responsible of controlling the relations over model components. Figure 4-22 shows the graphical representation for control features and Figure 4-23 shows its textual representation.

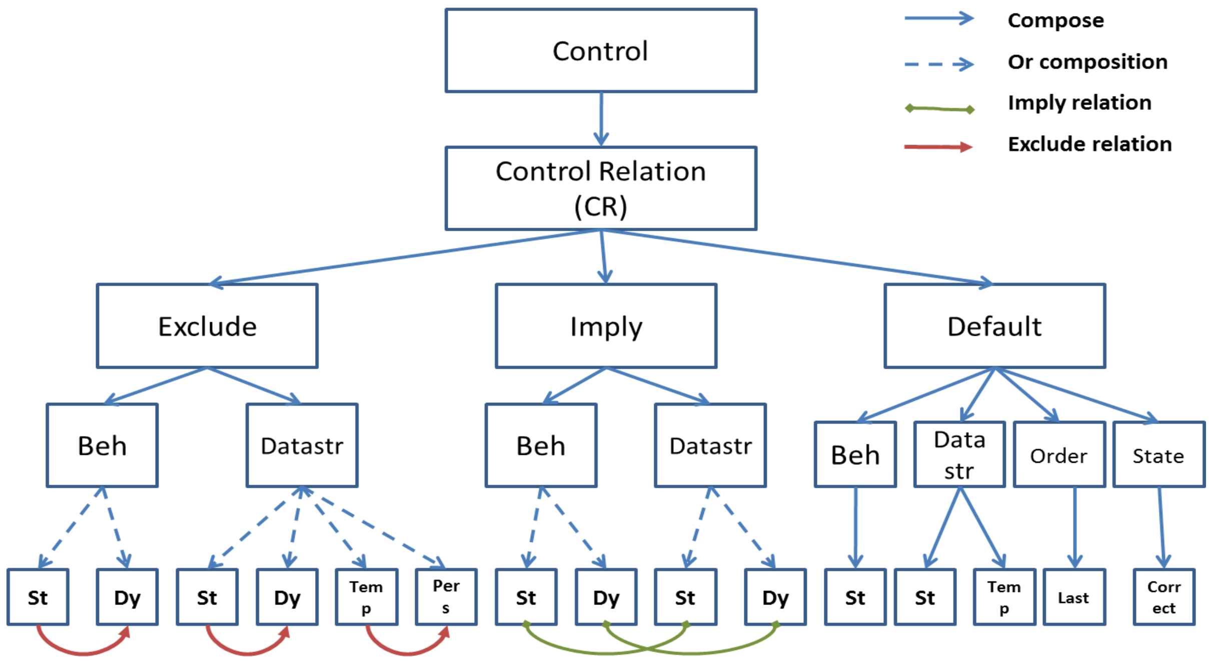


Figure 4-22 Set's Control Features (graphical representation)

```
Feature Control_Relation
{
Name: Control _Relation;
Type: CR;
Product Relation Exclude_1
{
Exclude_1.parts[1]=behavior.static;
Exclude_1.parts[2]=behavior.dynamic;
}
Product Relation Exclude_2
{
Exclude_2.parts[1]=datastr.static;
Exclude_2.parts[2]=datastr.dynamic;
}
Product Relation Exclude_3
{
Exclude_3.parts[1]=datastr.temporary;
Exclude_3.parts[2]=datastr.persistent;
}
}
```

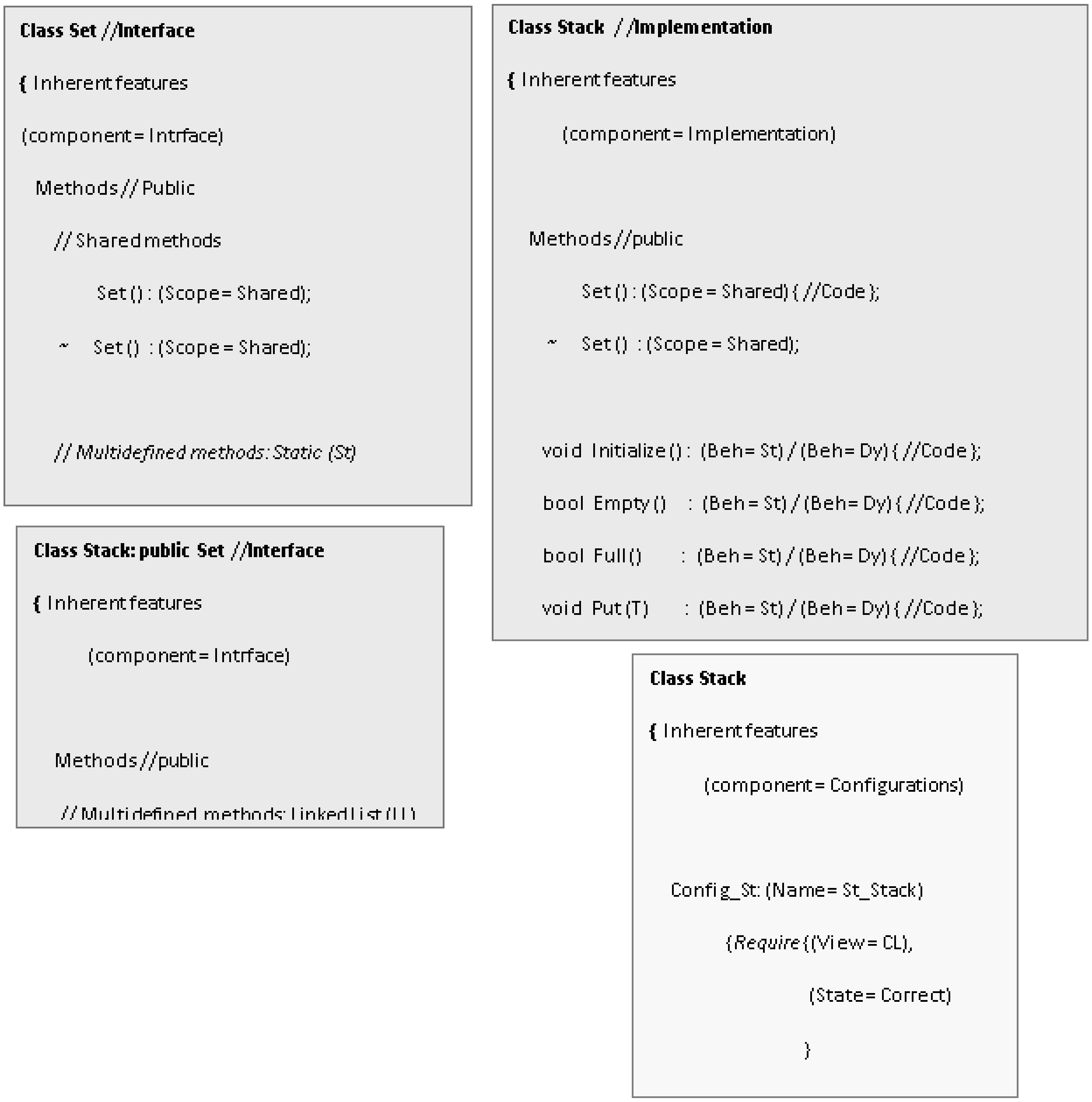
Figure 4-23 Set's Control Features (textual representation)

The last feature is in set's feature model is *Feature Configuration*. It stores the configured releases that had been done previously. For space reasons, we will show static stack configuration only. Figure 4-24 shows the textual representation for set configuration feature.

```
Features Configuration
{
Name: S_stack  configuration
Type: ConR;
Product Relation require
{
require.parts[1]=view.cl;
require.parts[2]=state.correct
}
Product Relation reject
{
reject.value=scope.shared;
reject.value=method.print;
}
}
```

Figure 4-24 Set Configuration feature

2- *Product Model:* Figure 4-25 shows the final product model for "Set" example. The figure specifies the "Set" interface, Stack sub-interface, Stack implementation, and a Stack configuration.



```
Class Set //Interface

{ Inherent features

(component = Intrface)

  Methods // Public

     // Shared methods

           Set () : (Scope = Shared);

     ~     Set ()  : (Scope = Shared);


     // Multidefined methods: Static (St)
```

```
Class Stack: public Set //Interface

{ Inherent features

        (component = Intrface)


   Methods //public

    // Multidefined methods: LinkedList (LL)
```

```
Class Stack  //Implementation

{ Inherent features

        (component = Implementation)


   Methods //public

           Set () : (Scope = Shared) { //Code };

     ~     Set ()  : (Scope = Shared);


     void  Initialize () :  (Beh = St) / (Beh = Dy) { //Code };

     bool  Empty ()    :  (Beh = St) / (Beh = Dy) { //Code };

     bool  Full ()     :  (Beh = St) / (Beh = Dy) { //Code };

     void  Put (T)     :  (Beh = St) / (Beh = Dy) { //Code };
```

```
Class Stack

{ Inherent features

        (component = Configurations)


   Config_St: (Name = St_Stack)

          {Require {(View = CL),

                    (State = Correct)

                   }
```

Figure 4-25 "Set" Product Model

4.6 Discussion

We started this chapter with a textual SPL design methodology (TDM) showing ordered steps for designing and implementing software.

Feature modelling approach was defined in the second section showing our feature model. This model is composed by *Features Meta-Model* that categorizes features into four categories, and a Product Model that captures system variabilities.

Class model was reported in the third section showing our enhancement on object-oriented class model by allowing versions of attribute definitions and method implementations.

Connecting feature model with class model was reported in the fourth section.

We closed this chapter presenting a model instance (Set example) to show the real implementation for our approach.

CHAPTER FIVE: IMPLEMENTATION ISSUES, EVALUATION AND APPLICATION AREAS

5.1 Introduction

In this chapter, we will show the implementation issues for our work. We will discuss the application issues and areas where TDM can be used. We will evaluate our work by presenting the concepts that we modify in conventional approaches and define our own concepts, and compare the work and results we obtained with other works. We will end this chapter by a conclusion describing the perspectives and future works.

5.2 Implementation issues

The implementation environment of this methodology requires a strongly typed and an object-oriented programming language. The checking process should guarantee the correct association between the Meta-Features model, Features Meta Model, Features Types, Features Global, Features Control, Features Configurations, Product Meta Model, and Product Model. We needed to add an extension to an existing OOPL, to adopt the concepts of TDM.

We are building on extensions that were presented earlier in (S. Ghoul, 2011). These extensions might be processed by any OOPL pre-processor. Configurations can be created according to its Feature model.

5.3 Application areas

Software engineering process will be strengthened by adding TDM to its feature modelling techniques, since it is more natural than current conventional approaches in presenting features and classifying them.

Our approach in highly recommended to be used in any feature modelling area like configuration management, feature-oriented programming, product family engineering and software product lines.

Real examples for real systems that may use TDM in their programming is operating system implementation, multi-agent systems and any system that needs feature separation and classification.

5.4 Evaluation

In the following, we will compare the power of our concepts (new or enhancement of old ones) and compare our contributions with similar works.

TDM Concepts vs. conventional related concepts: the following table summarizes the comparison of TDM concepts with others related ones.

Table 5-1 TDM Concepts vs. conventional related approaches

Concept	Current approaches	Our approach
Features Meta-Mode	Conventional approaches like (Gunther and Sunkle, 2012; , 2010; Kacper et al., 2011; Laguna and Marques, 2009) and other research works have described meta-model in term of features that have more than one sub-features as children.	Features Meta-Model is a structure that captures system's most characteristics. This meta-model is composed by: *Features Types, Features Global, Features Control* and *Features Configuration*. The relations between these meta-model features as specified.
Features Types	Each feature is defined individually. No support for full declaration for systems' features.	We present the type of all features, their possible values and relation declaration in the composed feature Features Types. It defines all acceptable cases for the features that construct system's variability.
Features Global	Shared features are not separated as a unit, but defined in the feature diagram hierarchy.	Global features are a sub-set from the Features Types. It defines the shared features for all system's components. They are modelled as a separated model unit.

Features Control	Relations between features are not separated as a unit, but defined along with features.	Control features are relations that specify coherence of configuration. They are presented as a separated model unit.
Features Configuration	Configurations represent a set of selected features from features model that specify a unique release or system version. They are presented along with the feature model.	Configuration features contains relation between features composing a release. Features Configuration contains all the product releases.
Relation	Conventional approaches describe the relation as a constraint between two or more components that have to be reserved.	Relation is a feature that may have several values and types. We did this to enhance system's tractability and maintenance. Since dealing with relations as features shows them in a structural and clear way and makes adding relation process more systematic.
Product Interface	Conventional approaches are weak in support component's interfaces. Each class is created based on its configuration characteristics.	We created a component interface to increase the modularization by separating the main concerns from the concrete components.

Comparison with similar works: while reading the literature (Gomaa and Shin, 2008; Istoan, 2013; Jézéquel, 2012; Laguna and Crespo, 2012), we found that feature modeling approaches can be compared based on several criteria. We selected the most recent and closest researches to our work, and we choose the most important (from our vision) comparison criteria to be:

1. Covered steps in software process.
2. Providing concise notation for feature modelling and meta-modelling.

3. Providing concise notation for class modelling.
4. Allow mixing feature-model and class models.
5. Use minimal number of concepts and have a uniform semantics.
6. Allow variability modelling.
7. Supporting methodology

And the papers used in this comparison are:

A. Clafer: unified modelling language (Kacper, 2010), Features and meta-model in clafer (Kacper et al., 2011).
B. OOFM- A feature modelling approach to implement MPLs and DSPLs (Sarinho and Apolinario, 2010).
C. A text based approach to feature modelling (Classen et al., 2011).
D. rbFeatures: feature-oriented programming with ruby (Gunther and Sunkle, 2012).
E. FAMILIAR: a domain-specific language for large scale management of feature models (Acher et al., 2013).
F. Our model.

In the following subsections, we present the detailed comparison.

Covered steps in software process.

Generally languages are specific to a given step in the software life cycle. These steps are: Requirements analysis, software architecture, design and implementation.

In (Kacper, 2010), the author presented CLAFER as a unified language for class and feature modelling. So he covered the design and implementation step very well. Sarinho et al. (2010) tried to combine object oriented concepts with feature modelling concepts to produce object oriented feature modelling. They clearly covered the implementation step but the design step was slightly represented. An implementation-only model was reported in (Classen et al., 2011) to present a textual feature modelling language. This step and design step were presented in (Gunther and Sunkle, 2012) too in a very tidy way to show the feature model over a dynamic programming language “ruby”. Acher et al. (2013) presented FAMILIAR language that covers the requirement analysis step with a great implementation description for feature and class models.

In our work, we covered design and implementation steps by providing a TDM based on *Features Meta-Model*, Feature Model, Product Meta-Model and Product Model, and enhancing the class model.

Provide concise notation for feature modelling and meta-modelling.

Kacper (2010) presented the feature model as a set of type definitions, features and constrains. And his work has been extended in (Kacper et al., 2011) to handle feature and meta- model.

Sarinho et al. (2012) proposed feature model as a part of object oriented feature model profile. There was no clear graphical notation to explicitly define the feature model. They used *FeatureTypes*, *AttributeTypes*, *GroupTypes* and *Constraints* to build the OOFM profile.

Classen et al. (2011) presented a textual feature model with feature declaration and hierarchy. This model is composed by features' attributes, constraints and structure. There was no mention for features' meta- model. In contrast to this, Gunther and Sunkle in their work (Gunther and Sunkle, 2012) presented a well and very clear feature model including meta-model and very tidy notation to present systems variation points and their relations.

Acher et al. (2013) decided to use textual scripting language instead of graphical notations to present their feature model. However, this does not prevent them to use the graphical notation in some small parts of their models.

In our work, a clear Feature Model including Feature Meta-Model was presented in chapter 4 showing four main categories for capturing any system features. The Meta-Model provides a features design pattern which may be used in designing any variable product.

Provide concise notation for class modelling.

Class modelling is a very significant phase. It provides a good description for system's structure and specifies the relations between its classes. Kacper in (Kacper, 2010; Kacper et al., 2011) gave a brief description for this model with a “telematics system”. This example was very poor with semantic that describes the system functionality.

Sarinho et al. (2012) presented feature class model as the main component of the object oriented feature model they propose. The authors reported that their class model shows different type of attributes and methods.

Classen et al. (2011) and Acher et al. (2013) works did not mention class models directly. They only focused on feature models. Even Acher et al. in their work (Acher et al., 2013) described all class's components (attributes, methods, values and types, encapsulation), but they didn't specify a concise notation for class modelling. In contrast to this, Gunther and Sunkle in their work (Gunther and Sunkle, 2012) presented class models in a very good way (class numbers, class operations …..).

In our work, we enhance the object oriented paradigm by providing generalized view for attributes and methods versions. Each attribute or method may have several versions that reflect the domain features.

Allow mixing feature-model and class models.

Mixing between feature model and class model presents a new model that consists of object oriented and features model together. Kacper (2010) did not support this in his work. But a map between feature model and class model was presented by adding constraints to the feature model to show class model.

Sarinho et al. in their work (2012) mixed class model and feature model in object oriented feature model profile with all OO relations and resources. This step was missing in the work reported in (Classen et al., 2011) and (Acher et al., 2013), since they don’t have class model representation. Gunther and Sunkle (2012) presented a class model separately from feature model. There was no mixing. In our work, mixing feature model with class model was supporting by extending class concepts with feature concepts. Each class interface, class implementation, attribute, and method possesses a set of features allowing its selection in a configuration.

Use minimal number of concepts and have a uniform semantics.

Using minimal number of concepts to represent the system and having a uniform semantics is a very powerful point for any proposed language. It minimizes the system complexity and confusion. Kacper in his works (Kacper, 2010; Kacper et al., 2011) didn't mention this point during description CLAFER. Each case was presented separately with its syntax and semantic.

Sarinho et al. (2012) used object constraint language to present the object oriented feature model semantics supplied with a very good example representation. Classen et al. (2011) presented a very strong semantic for their feature model that exceeds original feature model semantics. But they didn't show any effort to minimize these notation and concepts to have a uniform semantic.

Gunther and Sunkle (2012) tried to minimize concepts by decomposing feature model into several components like first-class entities (feature, feature model, product line, product variant and validations), helper entity and other components that format a uniform semantic for their rbFeatures model. Acher et al. (2013) did not presented a clear semantic representation. Even the authors tried to represent their language's constructs like (values and types, storage and variables… etc.) but the concepts were scattered and does provide a uniform semantic.

We extend the OO model only by supporting variability at attribute and method level. This variability was supported attaching features to attributes and methods versions. the feature meta model, the features model, and the product meta model are only as ordinary required for syntax and semantics analysis.

Allow variability modelling.

Variability modelling provides a deep view of concrete features (values) for feature model that system may contain and present the physical model for the system. It is very clear that Kacper in his work (Kacper, 2010; Kacper et al., 2011) did not support this kind of modelling since the last model was presented is class model.

Sarinho et al. (2012) approach mixed variability and class model in the term of feature modelling package, that contains all resources and relations over the entire system. TVL model that reported by Classen et al. (2011) didn't provide a direct variability model. It is hidden in the feature model they presented in their work. This is in contrast to the work reported in (Gunther and Sunkle, 2012), where variability model was clearly and strongly presented using UML notations and formally specified its syntax and semantics. Acher et al. (2013) presented this model textually without a uniform pattern of syntax to show the variation point and its possible values.

In our work, we believe that variabilities are important to show systems' possible cases and functionalities. So we provide with the feature model a variability model representation to show the possible values for each feature.

Supporting methodology

None of the presented works have developed a design and implementation methodology for software configuration. Configurations were done individually without any formal way.

In our work, a textual design and implementation methodology was presented shown ordered steps for software configuration.

This comparison is summarized in Table 5-2 Comparisons with other's work). The comparison criteria are numbered from 1-7 and the papers used for the comparison are from A-G. ✓ Symbol means strongly supported. ✗ Symbol means not supported, and ◆ symbol means weak supporting.

Table 5-2 Comparisons with other's work

Paper/ Criterion	**Clafer**	**OOFM**	**Classen et al.**	**rbFeatures**	**FAMILIAR**	**Our approach**
1	✓	✓	◆	◆	◆	✓
2	✓	✗	✗	✓	✓	✓
3	✗	✓	✗	✓	✗	◆
4	✗	✓	✗.	◆	✗.	✓
5	✗	✓	✗	◆	◆	✓
6	✗	◆	✗	✓	✓	✓
7	✗	✗	✗	✗	✗	✓

5.5 Conclusion: perspectives and future works

Through our study about feature modeling and SPL engineering, we found that current feature models did not support feature modularization and separation. We found that linking feature models with class models is still weak and does not reflects feature model's concepts, and there is a lack in variability design and implementation methodology.

In section 1.5, we proposed 4 contributions to be done during this thesis. The first contribution was clearly done through the textual feature design methodology that supports software product line engineering. The second contribution was done by defining four meta-feature models to support feature modularization and to classify features based on their functionalities.

The third contribution was done by enhancing object-oriented class model and formally defined the links between feature model and class model to allow mixing features' concepts with real implementation for classes. And finally, the last contribution was done by proposing a procedure for configuration generation based on pre-selected features.

Our work can be extended and developed in future to:

- TDM enhancement and evolution.
- Define other meta-feature models to capture all software's variability features.
- Enhance current class model to be more realistic and reflects feature model in uniform and formal way.
- Enhance the configuration generation to be a smart automated generation.
- Design a uniform language mixing features and classes.

References

Acher M., P. Collet, P. Lahire & R. B. France, (2013). FAMILIAR: A domain-specific language for large scale management of feature models. *Science of Computer Programming. (*78): 657-681.

Apel S., A. von Rhein, T. Thum & C. Kustner, (2013). Feature-interaction detection based on feature-based specifications. *Computer Networks.*

Arnaud H., B. Quentin, H. Herman, Rapha, M. l & H. Patrick, (2011). Evaluating a textual feature modelling language: four industrial case studies, Proceedings of the Third international conference on Software language engineering, 337-356.

Asikainen T., T. Mannisto & T. Soininen, (2006). A unified conceptual foundation for feature modelling, 10th International Conference of Software Product Line 31-40.

Classen A., Q. Boucher & P. Heymans, (2010). A text-based approach to feature modelling: Syntax and semantics of TVL. *Science of Computer Programming. (*76): 1130-1143.

Classen A., Q. Boucher & P. Heymans, (2011). A text-based approach to feature modelling: Syntax and semantics of TVL. *Science of Computer Programming. (*76): 1130-1143.

Don B., (2005). Feature models, grammars, and propositional formulas, Proceedings of the 9th international conference on Software Product Lines, 7-20.

Ghoul S., (2011). Supporting Aspect-Oriented Paradigm by bio-inspired concepts, IEEE explorer, Nov. 29 2011-Dec. 1 2011, 63-73.

Gomaa H. & M. E. Shin, (2008). Multiple-view modelling and meta-modelling of software product lines. *Software, IET. (*2): 94-122.

Gunther S. & S. Sunkle, (2012). rbFeatures: Feature-oriented programming with Ruby. *Science of Computer Programming. (*77): 152-173.

Istoan P. 2013. Methodology for the derivation of product behavior in a Software product Line. Ph.D thesis. Universit e de Rennes 1, University of Luxembourg (LASSY).

Jézéquel J.-M., (2012). Model-Driven Engineering for Software Product Lines. 24 Sep 2012. *Report*.

Kacper B., (2010). Clafer: a unifed language for class and feature modeling. April, 2010. *Report*.

Kacper B., C. Krzysztof & W. Andrzej, (2011). Feature and meta-models in Clafer: mixed, specialized, and coupled, Proceedings of the Third international conference on Software language engineering, 102-122

Laguna M. A. & J. M. Marques, (2009). Feature Diagrams and their Transformations: An Extensible Meta-model, 35th Euromicro Conference on Software Engineering and Advanced Applications., 27-29 Aug. 2009, 97-104.

Laguna M. A. & Y. Crespo, (2012). A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring. *Science of Computer Programming.*

Marcilio M., B. Moises & C. Donald, (2009). S.P.L.O.T.: software product lines online tools, Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications, 761-762.

Marco S. & D. Sybren, (2007). Classifying variability modeling techniques. *Information and Software Technology (*49): 717-739.

Razieh B., N. Shiva, Y. Tao, G. Arnaud & B. Lionel, (2012). Model-based automated and guided configuration of embedded software systems, Proceedings of the 8th European conference on Modelling Foundations and Applications, 226-243.

Reiser M.-O., (2009). Core concepts of the compositional variability management framework (CVM): A practitioner's guide. *Report*.

Sarinho V. T. & A. L. Apolinario, (2010). Combining feature modeling and Object Oriented concepts to manage the software variability, IEEE International Conference on Information Reuse and Integration (IRI), 4-6 Aug. 2010, 344-349.

Sarinho V. T., A. L. Apolinario & E. S. de Almeida, (2012). OOFM - A feature modeling approach to implement MPLs and DSPLs, IEEE 13th International Conference on Information Reuse and Integration (IRI). 8-10 Aug. 2012, 740-742.

Savinov A., (2012). Concept-oriented programming: classes and inheritance revisited, 7th International Conference on Software Paradigm Trends (ICSOFT 2012), 12, 24-27 July 2012, 381-387.

Sim-Hui T., (2013). Problems of Inheritance at Java Inner Class. *ArXiv e-prints.*

Stephan M. & M. Antkiewicz, (2008). Ecore.fmp: A tool for editing and instantiating class models as feature models. 05/2008. *Report*.

Teixeira L., P. Borba & R. Gheyi, (2011). Safe Composition of Configuration Knowledge-Based Software Product Lines, 25th Brazilian Symposium on Software Engineering (SBES). 28-30 Sept. 2011, 263-272.

Thaum T., C. Kustner, F. Benduhn, J. Meinicke, G. Saake & T. Leich, (2012). FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming , Available online 21 June 2012.*

ملخص

تصميم خط إنتاج البرمجيات مهم جدا لزيادة عملية إعادة استخدام النظام وتقليل التكاليف والجهود لبناء مكونات من الصفر لكل برامج التكوين.

عدة نهج سابقة تعاملت مع عملية الهندسة باستخدام العديد من التقنيات. وقد كان أشهرها تقنية فصل القواسم المشتركة والتنوع لمكونات النظام مما يتيح السماح باختيار التكوين على أساس الخصائص التي يحددها المستخدم. هذه النهج تعاملت مع جميع مراحل تطوير البرمجيات، ولكن المراحل الاكثر تحدي هي التصميم والتنفيذ.

النهج النصية المستندة إلى تدوين تم استخدامها بسبب التمثيل الرسمي لمعانيها ودلالاتها المستخدمه لتمثيل ميزات النظام والتطبيقات. ولكن هذه الأساليب لا تزال ضعيفة في ملامح خلط (المستوى المفاهيمي) والطبقات (المستوى المادي) والتي تضمن سلاسة وتلقائية جيل التكوين لإصدارات البرامج.

في هذه الأطروحة، ونحن سوف يعزز عملية صناعة خط انتاج البرمجيات من خلال تعريف فوقية الميزات التي تحتوي أهم خصائص مفاهيم النمذجة الميزة، وتصنيف هذه الميزات وفقا لوظائفها. وسوف نسمح لحدوث عملية خلط المفاهيم والميزات في طريقة بسيطة باستخدام واجهات التطبيقات والميزات الملازمة للتحرك على نحو سلس من نموذج إلى نموذج ميزة فئة.

عملية صناعة خط البرمجيات سيتم إثراءها مع تصميم وتنفيذ منهجية نصية تخلط النموذج وميزة في طريقة جديدة. وتتيح هذه المنهجية تعريف نموذج الطبقة ابطريقة تعكس ملامح نموذج المفاهيم مع خلط متسقة مع نموذج الميزة. لأنه يعزز عملية تكوين جيلالبرمجيات لان يكون أكثر بساطة، وأكثر تماسكا وكاملة.

نموذج خطي لتصميم خطوط انتاج البرمجيات يجمع بين خصائص الفئات والمميزات

بواسـطة
علا عبد الرؤوف يونس

بإشــراف
أ.د. سـعيد الغول

قدمت هذه الرسالة استكمالاً لـمـتطلبات الحصول على درجة الـمـاجـسـتير في عـلـم الـحـاسـوب

عـمـادة البحث العلمي والدراسات العليا
جامعة فيلادلفيا

حزيران 2013