Chapter 4

Feature-Oriented Programming in Model Driven Development

In this chapter a focus is given on the technical aspects of FOP by modifying the reusability concepts through interfaces, objects and lifters. Lifters are generalization of inheritance comprising the reusability of features. It has been proposed to use lifter through FOP in order to handle an array. This proposed method of reusability supports the concept that FOP is not only an incremental software development tool but also can be used in software maintenance and in software up gradations. In order to make FOP popular, an attempt has been made by presenting the appropriate definitions of features. Apart from that, steps for handling FOP and the necessity of FOP in product line have been given. As a part of handling FOP, pseudo codes are proposed. The pseudo codes proposed in this work will be helpful in converting to a feature-oriented source code. The necessity of FOP in MDD has been demonstrated through a case study.

4.1 Background

Software up gradation is required due changes frequent to in requirements of the users. Hence software should possess the characteristics of adaptability, extensibility and customizability throughout its life cycle. The process of continuous adaptation and customization is known as software evolution. It has been found that software evolution can be handled in a more efficient way by reusing the existing features of the software. Hence the ideal goal of software engineers should be to reuse from previous versions, software to build a new version of the as much as possible. However, reuse of existing codes involves a major difficulty. The more code one reuses, the less it fits in different contexts of deployment. This has motivated researchers to
find an alternative for better software evolution. As a result, a new area of software development emerged, known as Software Product Lines (SPL).

Software product lines/program families promise a solution by generating families of programs and not by generating one single monolithic piece of software. *Software product lines* refer to engineering techniques for creating a portfolio of similar software systems from shared sets of software assets using a common means of production. Product line architecture [81] supports in building families of systems from components. It refers to software engineering methods, tools, and techniques for creating a collection of similar software systems from a shared set of software assets using common means of production.

A product line is used by two different categories of persons. One who generates a program (known as deployer) and the other who implements a program. The deployer creates a program by choosing from a set of features. However, one who implements the program has a major role in Product Line. This person designs the entire software by identifying the common set of software, integrating with the unshared components and resulting in new software of the family. Usually, many combinations of features are allowed, resulting in a high variance of the generated programs.

Feature-oriented programming (FOP) [52] assists in designing and implementing the modular system product lines. The key idea is to arrange the design and implementation as a layered stack of functionalities where the layers consist of different programs. Thus, implemented grained layered architecture leads to reusable, extensible and customizable fine software. Motivated by these observations several studies have been made in the domains of databases [49] middleware, [50] avionics, [51] and network protocols [49] which show that FOP is an appropriate tool to implement such layered, step refined architectures.

### 4.2 Concepts of FOP

*Feature-oriented programming* (FOP) is a programming paradigm, where *Feature* is a major component. It is a tool for designing and implementing Software Product Line.
Feature, SPL and FOP are all inter-related to each other. Feature is a major component of FOP paradigm since it also plays a crucial role in feature modeling where features are used to describe the variability of a SPL. On the other hand, in FOP, feature is a first-class language construct for structuring source code. Hence, we need to explore the following.

a) Feature

b) Product line Architecture

c) FOP

4.2.1 Feature

A feature is a unit of functionality of a software system that satisfies a requirement, represents a design decision, and provides a potential configuration option. Typically, from a set of features, many different software systems can be generated that share common features and differ in other features. There are several definitions of a feature given by different researchers which are arranged below. The hierarchy of which extends from abstract to technical, in chronological order. This has been done for a better understanding of feature [103].

1. Kang, Cohen, Hess, Novak and Peterson [55]: “A prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems.”

2. Kang et al. [56]: “A distinctively identifiable functional abstraction that must be implemented, tested, delivered, and maintained.”

3. Czarnecki and Eisenecker [57]: “A distinguishable characteristic of a concept (e.g., system, component, and so on) that is relevant to some stakeholder of the concept.”

4. Bosch [58]: “A logical unit of behaviour specified by a set of functional and non-functional requirements.”

5. Chen, Zhang, Zhao and Mei [59]: “A product characteristic from user or customer views, which essentially consists of a cohesive set of individual requirements.”

6. Batory, Sarvela and Rauschmayer [52]: “A product characteristic that is used in distinguishing programs within a family of related programs.”
7. Classen, Heymans and Schobbens [60]: “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.”

8. Zave [61]: “An optional or incremental unit of functionality.”


10. Apel, Lengauer, Möller and Kästner [63]: “A structure that extends and modifies the structure of a given program in order to satisfy a stakeholder’s requirement, to implement and encapsulate a design decision, and to offer a configuration option.”

The first seven definitions reflect that features are abstract concepts of the target domain used to specify and distinguish software systems requirements, whereas the last three definitions depict that features must be implemented in order to satisfy requirements [103]. The switch over from requirement view to implementation view is due to the different uses of features.

The distinction between problem domain and solution domain [57], as illustrated in Figure 4.1, is useful for the classification of the definitions presented above.

![Figure 4.1: Problem and Solution domain](image)

The problem domain comprises concepts that describe the requirements on a software system and its intended behavior. The solution domain comprises concepts that define how the requirements are satisfied and how the intended behavior is implemented. The first seven definitions describe the feature concept from the perspective of the problem domain. Here, features are used to describe the expectations from a software system. The last three definitions describe features from the perspective of the solution domain, i.e., they show how features provide/implement the desired functionality [103].
4.2.2 Product Line Architecture

The set of software systems generated from a set of features is called a *Software Product Line* [53, 54]. Product line architecture is software architecture for a family of software products. In the software product family, individual products share common parts and functionality, but some parts are different and need to be customized. Product line architecture defines the concepts, structure and texture that are necessary to achieve variation in features of variant products while achieving maximum sharing of parts during the implementation.

4.2.3 Feature-Oriented Programming

Feature-Oriented Programming (FOP) is a design methodology and a tool for program synthesis. The goal is to specify a target program declaratively in terms of the features that it offers and to synthesize an efficient implementation that meets these specifications. FOP studies the modularity of *features* in product lines, where a feature is an increment in program functionality [52]. Hence, the idea of FOP is to synthesize software (individual programs) by composing features. Typically, features refine the content of other features in an incremental fashion. Hence, the term *refinement* refers to the set of changes a feature applies to a base code. Adding features incrementally, called *stepwise refinement*, lead to conceptually layered software designs.

4.3 Implementation of Features

Abstracting programs in terms of features facilitates the implementation of product lines. Conventional modularization approaches like class, packages are not appropriate for feature modules. Here, a typical feature implementation spreads over several module (class, package) boundaries. Further, a single module (class, package) may contain fragments from multiple features. To achieve such encapsulation of fragmented features, many mechanisms are available like mixin layers [52, 63], lifters [93] etc.

4.3.1 Mixin Layer Approach

*Mixin layer* [52, 63] is one of the approaches of implementing features. The basic idea is that features are seldom implemented by single classes. Typically, a feature
implements collaboration, which is a collection of roles represented by mixins that cooperate to achieve an increment in program functionality. FOP aims at abstracting and explicitly representing such collaborations. A mixin layer is a module that encapsulates fragments of several different classes (roles) so that all fragments are composed consistently. Figure 4.2 depicts a stack of three mixin layers (L1 − L3) in top down order.

![Stack of three mixin layers](image)

**Figure 4.2:** Stack of three mixin layers

These mixin layers crosscut multiple classes (C_A − C_C). In Figure 4.2, white boxes represent classes or mixins; gray boxes denote the enclosing feature modules; filled arrows refer to *mixin-based inheritance* for composing mixins. The layers L1, L2 and L3 are the three features (mixin layers). Each feature embeds few classes like L1 (feature 1) embeds C_A and C_B classes, L2 (feature 2) embeds C_A, C_B and C_C, and L3 (feature 3) embeds C_B and C_C classes. Figure 4.2 also shows the inheritance of attributes between the classes of different mixin layers like L2 layer classes inherit some attributes from L1 classes and L3 layer classes inherit few attributes from L2 layer. This is depicted by the arrows. This inheritance is required for better composition of features and is known as mixin based inheritance.

### 4.3.2 Lifters Approach

Reusability is one of the major contributions of OOP in terms of inheritance. The inheritance approach of OOP has very limited capability in FOP. The reusability
property of OOP is further enhanced in FOP by mixins. These modifications of inheritance mechanism brought into existence of another concept of lifters. Lifters are better than inheritance as they allow taking multiple features at a time. This is a prerequisite of product line. Though mixin based inheritance is a very common approach but lifters are more powerful than mixin based inheritance in context of product line.

Lifter is similar to inheritance in context of method overwriting. But the differences are

- lifters depend on multiple features
- separate entities are used for composition
- they are used for feature interactions

Mixin based inheritance support reusability in FOP but they cannot handle feature interactions. On the other hand, lifters have no language support in some popular languages like FeatureC++ till date. However, this cannot be a drawback as it not only has powerful reusability mechanism but it also allows for feature interactions by selecting multiple features. This is a major contribution towards flexible composition of features. If it was integrated in popular high level languages then it could have replaced mixin based inheritance mechanism in FOP.

### 4.4 Implementation of FOP

FOP is a concept, which can be implemented in language level and also in product line architecture. The former is required for developing codes directly whereas the later is required in Model Driven Development (MDD). FeatureC++ has been explored in context of language level and in product line architecture.

#### 4.4.1 FOP in Language Level

The present research works in the programming languages are very vast and many FOP supporting languages are in development stage. One of the popular languages is FeatureC++ which supports Feature-oriented Programming for C++. This language is an enhancement of C++ in FOP context. Some of the important characteristics of FeatureC++ are as follows [104].
- A source file of FeatureC++ has an extension .fcc and comprises exactly one mixin class.
- Features are implemented by Mixin layers. Mixin Layers are represented by directories of the file system as preprocessor directives (included source files).
- While using the feature name, the directory search path is browsed to find the directory (Mixin Layer) and their corresponding Mixins.

Each mixin class consists of constants and refinements. Refinements introduce new attributes and methods, which can override methods of their parent classes. In order to access the overridden method super-keyword is used. Super refers to the type of the parent mixin. Hence it can be said that overridden methods through super keyword, refinements and mixin layers are the foundation of FeatureC++. Apart from these extensibility and constructor handling are also done by refinement in a very comprehensive way. Multiple inheritance, class and method templates and overloaded method propagation are all supported by FeatureC++. The grammatical rules in FeatureC++ are used for declaration of the Mixin Layers, refinements, inheritance etc. Table 4.1 summarizes few grammatical rules of FeatureC++ [104].

<table>
<thead>
<tr>
<th>Rules</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>layer &lt;layer name&gt;</td>
<td>declares the name of the enclosing Mixin Layer</td>
</tr>
<tr>
<td>refines &lt;class declaration&gt;</td>
<td>refines constants or refinements</td>
</tr>
<tr>
<td>super::</td>
<td>refers to the type of the parent Mixin</td>
</tr>
<tr>
<td>super::&lt;method name&gt;</td>
<td>invokes a method of the parent class</td>
</tr>
<tr>
<td>super::&lt;attribute name&gt;</td>
<td>accesses an attribute of the parent class</td>
</tr>
<tr>
<td>this::&lt;class name&gt;</td>
<td>refers to the class type of the current layer</td>
</tr>
</tbody>
</table>

The mixin layer declarations, referencing the parent mixin for accessing an attribute or method of the parent class etc. can be done by FeatureC++ but the mixin layers are able to control the static crosscutting concerns. Dynamic crosscutting concerns need to be considered by enhancing FeatureC++ ability. We also find that refinements can be done in a better way if the lifter concept is incorporated in language level.
4.4.2 Architectural Model of FOP

Architectural model of FOP is known as FOP domain model. It is based on algebra where constants and functions (i.e., operators) are composed to define a domain of programs that can be synthesized. As relational algebra is a set of operators, whose compositions define a domain of query evaluation programs, similarly a FOP domain model comprises a set of constants and functions whose composition yields a range of programs. It is an abstract way to specify building blocks of programs.

The algebraic domain model of FOP has also been represented by GenVoca [12]. This is based on step-wise refinement (SWR) which is a methodology for building software by progressively adding details. GenVoca extends step-wise refinement by scaling refinements to a component or multi-class granularity, so that applications of significant complexity can be expressed by composing a few large-scale refinements called feature refinements. GenVoca defines an architectural model for FOP.

4.5 Proposed Steps for Handling of FOP

Though FOP is a very powerful concept in SPL but the handling of FOP is not known to the user. Hence, this powerful concept has not gained much popularity. In order to make the FOP concept user friendly, the following major steps for working in FOP environment using lifters have been chalked out.

1) Define interfaces.
2) Create new features, for base implementations from the existing interfaces.
3) Create objects with all the features for interaction handling in a particular order.
4) Provide lifters to replace method over writing in subclasses.

Interfaces are defined to support several implementations. Furthermore, they ease translation into a high level language, like Java/FeatureC++, in which a class can implement several interfaces. Codes are written for base implementations of individual features. Objects are composed of several features in a particular order for interaction handling. Finally, the lifters are required to adapt the functions of an interface which is essential for flexible composition.
4.5.1 Proposed Array Handling Model

A model of array handling is proposed which provides some basic functions of array. The model is designed to introduce FOP with lifters. This model supports the reusability not through conventional mixins but with the new method known as lifters.

This model provides the following array operations in a **one dimensional** and a **two dimensional array**.

a) **Insert**: provides insertion of a fixed number of elements in any of the arrays.

b) **Traverse**: provides traversal (displaying) of all elements in the array.

The characteristics of the model are as follows.

(a) The insert/traverse operations are controlled by enable/disable (called as **Noperation**) operation.

(b) The number of elements is controlled by a mechanism, called **Size**.

(c) All the elements are considered as integer type.

4.5.2 Defining Interfaces & Features for the Proposed Model

In this section, the pseudo codes are formulated to represent the four interfaces, define the required **features**, and to declare the objects.

The model creates four interfaces. They are Onedimarray, Twodimarray, Size and Noperation. Each interface (Figure 4.3) has its own member methods.

```java
interface Onedimarray {
    void clear();
    void insert1(int);
    void traverse1();
}

interface Size {
    void reset();
    void input_one_size();
    void input_two_r_c_size();
    int get_one_size();
    int get_two_row_size();
    int get_two_col_size();
}

interface Twodimarray {
    void omit();
    void insert2(int, int);
    void traverse2(int, int);
}

interface Noperation {
    void noperations();
    void operations();
    boolean is_operation();
}
```

**Figure 4.3:** Interface Definition
The next step is to create new features from the existing interfaces. There are four features which are the following.

- **FA1** - implements the interface Onedimarray.
- **FA2** – implements the interface Twodimarray.
- **FS** - implements the interface Size.
- **FN** – implements the interface Noperation.

The features comprise the method descriptions of all the member methods of the interface, apart from own member data (Figure 4.4). All the implementations given below are the base implementations of individual features.

```java
feature FA1 implements Onedimarray {
    int s;
    void clear() {
        s=0; x=new int[s]; delete [] x; }
    void insert1(int t) {
        x=new int[t]; int i;
        for(i=0;i<t; i++) { input x[i]; }
    }
    void traverse1(int t) {
        int i;
        for(i=0;i<t;i++) { display x[i]; }
    }
}

feature FS implements Size {
    int p1= 0, p2=0, p3=0;
    void reset() { p1 =p2=p3= 0; }
    void input_one_size() {input p1; }
    void input_two_row_col_size() {
        input p2, p3; }
    int get_one_size() { return p1; }
    int get_two_row_size() { return p2; }
    int get_two_col_size() { return p3; }
}

feature FA2 implements Twodimarray {
    int r,c;
    void omit() { r=c=0; }
    int (* x)[r]=new int[r][c]; delete [] x; }
    void insert2(int t1,int t2) { int (* x)[r]=new int[r][c]; int i, j;
        for(i=0;i<t1; i++) {
            for(j=0;j<t2; j++) { input x[i][j]; }
        }
    }
    void traverse2(int t1,int t2) {
        for(i=0;i<t1; i++) {
            for(j=0;j<t2; j++) { display x[i][j]; }
        }
    }
}

feature FN implements Noperation {
    boolean g = true;
    void noperations() {
        g = false; }
    void operations() {
        g = true; }
    boolean is_operation() {
        return g; }
}
```

**Figure 4.4: Feature Definition**
The third step is to create object. Mainly, features are used as new constructors to create objects. While creating an object (Figure 4.5), multiple features are selected but one of the features is used as a new constructor. The other features are used with the constructor feature for implementations of those features. This is achieved by the following declarations.

\[
\begin{align*}
\text{new } FN & (FS (FA1)) \\
\text{new } FN & (FS (FA2)) \\
\end{align*}
\]

**Figure 4.5: Object Declaration**

In the above declaration, two objects have been created. The first object is responsible for handling a one dimensional array whereas the second object is responsible for handling a two dimensional array. In the first object, FA1 is used as a new constructor. In FA1, other two feature implementations i.e., FS and FN are used for creating an object with all three features. Similarly, in the second object, FA2 is used as a new constructor. FS and FN are used in as implementations. The interactions handling among the features are possible by combining the features in a particular order. Hence, the order of composition in creating the first object is FA1 which first adds FS to FA1 and then adds FN. Similar is the methodology for the second object.

### 4.5.3 Introduction of Lifters in the Proposed Model

The last step is to provide lifters which are replacements of method overwriting in subclasses. Lifter is a separate entity. In the present model, each lifter is capable of handling two features at a time. The final feature implementations take place by lifting features via interfaces.

Five lifters (figure 4.6) have been designed, with the following combinations of features and interfaces.

(a) FS feature has two lifters
   (i) Through interface Onedimarray
   (ii) Through interface Twodimarray

(b) FN feature has three lifters
(i) Through interface Onedimarray
(ii) Through interface Twodimarray
(iii) Through interface Size

Each lifter executes the codes of the parent method by using the super keyword; otherwise its own feature method is executed. This way the method overwriting is achieved. Moreover, the declaration feature FS lifts Onedimarray adapts the functions of Onedimarray to the context of FS. As feature FS is selected for object composition, so the lifter FS with interface Onedimarray does not implement Onedimarray directly but indirectly implements some methods of interface Size (this.method_name()) and some methods of interface Onedimarray (super.method_name()). The method overwriting takes place accordingly. The advantages of using lifter are.

(i) Easy method overwriting
(ii) Flexible composition of feature
Modules with individual services can be created as - FA1 and FA2 for one and two dimensional arrays respectively. Both modules have some common behavior like insert,
traverse with some minor variations. When insertion in one dimensional array is required a lifter with FS (a feature which implements interface Size) is composed through interface, Onedimarray. As FS feature can work for both one and two dimensional array in this case it adapts the functions of Onedimarray in this case. This flexible composition supports the idea that lifter can be an important component of FOP. This also helps to generalize OOP techniques and to give a new conceptual view to Model Driven Development (MDD).

4.6 Analysis of the Proposed Model

FOP has several advantages over any other modular programming. The modeling of array using FOP also reflects the following advantages.

(a) More flexibility is obtained, as objects with individual services can be composed from a set of features. The new constructor has created two different objects with three features in each. When more number of features are present, different objects can be created with different combinations of features.

(b) As the core functionality is separated from interaction handling, it provides more clarity on the dependencies among the features.

(c) The model proposes liftings or interactions between two features at a time and hence it is a very simple. Modified lifters may be proposed in case of dependencies among several features.

4.6.1 Utility of the Proposed Model

The composition of features in the above FOP technique assumes that features are composed in a particular order. Only from external interface, it is possible to view an object as composed of a set of features. Ordering with lifters helps in modifying the concepts of inheritance of OOP. Although inheritance is a very powerful mechanism for reusability, its use is restricted to OOP. Hence, inheritance by lifters is refined to support feature interactions. In contrast, features can be reused by simply selecting the desired ones during creating of an object.
Mixins have been proposed as a basic concept for modeling other inheritance concepts [52, 63]. If mixins are used then the composition of the features without lifters has to be done in layered structure. Designing in layered structure is more tedious and is error prone. Instead by using lifters, features can just be selected for software development.

4.6.2 Relevance of Array handling in Software Product line

Non-software products can be generated by features. A very common example is the Dell websites where HTML pages provide simple declarative domain-specific languages (DSLs) for specifying customized configurations of PCs (e.g., single or dual processor options, monitor options, disk options, etc.) [18]. An objective of FOP is to do the same for software products. So, the array model with features has been proposed to justify that software product lines can be also designed in a small scale for implementation of FOP.

FOP architectural domain model is needed for generating metaprograms, metaprogramming and meta-expressions. Metaprograms represent and manipulate other programs or themselves. The techniques used to represent and manipulate other programs or themselves are known as metaprogramming [78]. The meta-expressions are obtained through metaprograms which give an algebraic representation of a program. In Model Driven Architecture (MDA) metaprogramming plays a major role.

The array handling can be represented in architectural model using FOP which can achieve the meta-expressions. This array model shall be illustrating a software product line in small traditional programs. In Chapter 8, the array handling features as a software product line has been presented.

The next section deals with the case study on STNPL, which has been introduced in Chapter 3 (with coverage on security aspect) to illustrate the role of features in such designing.
4.7 Case Study: Smart Telephone Network Product Line

An authorized person gets access to the telephone network. After getting access, the person gets connected to the server for call processing. This needs some additional activities to be carried out. It is planned to design the entire process in terms of features rather than carrying out traditional designing. Feature-oriented approach of designing helps to arrange the activities in layered structure which gives a better understanding of the sequence of activities. It also allows adding activities in an incremental way, through layers. The components of the Telephone Network are now described.

4.7.1 Components of STNPL

The telephone network needs a few items to handle the call operations. As described in Chapter 3, this phase of designing is concerned about the call processing of the user, who has been authenticated to access the network. The various components required are.

a) Public Switch Telephone Network
b) Gateway
c) Checker
d) Server

These items need software as well as hardware devices. In the present work designing has been restricted to the software artifacts. The functions of the four components are as follow.

a) PSTN: It fetches, the security code and the time from the security aspect module and passes it to the gateway.
b) Gateway: This is an interface between the network and the server.
c) Checker: The interface uses this component to detect a busy or free line.
d) Server: The control goes to the server only after a detection of free line. The last phase of the call processing is done by the server.

4.7.2 Functioning of STNPL

The functioning of the Telephone network is depicted in the following sequence diagram.
PSTN is the base feature. The base feature also has a gateway. The main job of the base feature is to fetch the security number and the time. These are obtained by two
methods fetch security_no() and fetch time(). This information is passed to the gateway which is then again forwarded to the checker component for detecting a free line. On detection of a busy line, a message is given which then gets back to the gateway. This process continues until a free line is detected. Then the control is passed to the server. A connection is created between the server and the user. On receiving a Welcome message, the user provides the details of the call through the options provided by the server. The receive() completes the voice communication between the user and the destination. The callend() terminates the link between the user and the destination which is then followed by an execution of the method show_time() to display the call duration. Finally, the link between the user and the server is disconnected.

4.7.3 Handling of features in STNPL

The traditional OOP cannot declare objects with a combination of different features which are permitted by FOP. Hence the interfaces, features and object to implement the features are defined in this section.

The three interfaces are: (a) PSTN (b) Gateway (c) Server

It is hence required to present the pseudo code to the three interfaces, define the required features and declare the object.

The three interfaces are PSTN, Gateway and Server. The interfaces with their respective members are given in Figure 4.8.

<table>
<thead>
<tr>
<th>Interface PSTN</th>
<th>Interface Gateway</th>
<th>Interface Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>int fetch_securityno();</td>
<td>void accept_securityno();</td>
<td>boolean check_route();</td>
</tr>
<tr>
<td>float fetch_time();</td>
<td>void accept_time();</td>
<td>void receive();</td>
</tr>
<tr>
<td>}</td>
<td></td>
<td>void callend();</td>
</tr>
</tbody>
</table>

Figure 4.8: Interface Definition of STNPL

As the next step is to create features from the existing interfaces, hence the following three features are declared.

FP – implements the interface PSTN

FG – implements the interface Gateway

FS - implements the interface Server
Figure 4.9 gives the feature definition.

The following definition is of an object.

```
new FS ( FG (FP))
```

The object created will be responsible for handling the telephone network as per the functioning given in section 4.7.2.

### 4.7.4 Features vs. objects

The major objective of using features is that it increases the flexibility. The designing of the features, interfaces etc. rather than traditional objects clearly indicates that a better
clarity is obtained by segregating the functions first in interfaces like PSTN, Gateway etc. and then in definitions in the respective features.

The only limitation is that STNPL can be more robust if the server is capable of handling International and National calls with storage of the entire call details including the recording of the conversations. For the sake of simplicity only the call duration recording has been considered. This can be further used for billing purpose.

4.8 Summary

In this chapter the steps required for FOP handling using lifters have been suggested. An array handling model for demonstrating the steps has also been designed. The proposed model is helpful in designing a small-scale software product line. It has been observed that FOP can be implemented in language level as well as in architectural domain model. In language level it has been identified that lifter mechanism is lacking. Integration of lifters in a popular high level language like FeatureC++ has been suggested. In order to make FOP an important tool of MDD the FOP modules of STNPL has been designed. This designing highlights the advantages of FOP in this sophisticated product line. This supports the hypothesis that FOP can increase the efficiency of MDD techniques.