Chapter 7

Modular Refactoring

In this chapter, the role of Unified Modeling Language (UML) diagrams and Object Constraint Language (OCL) expressions in modular refactoring have been explained. It has been found necessary to investigate the refactoring of class diagrams. The modular refactorings, like class diagrams are done in the proposed STNPL. Modular refactoring has also been done on an existing module comprising codes. This is done to analyze the processes and suitability of integrating such modules in any existing software. As a part of such modular refactoring, the aspect modules are investigated in context of refactoring through RFDs. The role of UML and OCL has been explored for better visualization of modular refactoring.

7.1 Introduction to Modular Refactoring

Modular refactoring is a kind of model transformation [30] which is equivalent of program refactoring at model level, so that models are refactored instead of programs by preserving the behavior. Two well known model transformations are refinements and model refactoring. Refinements are done during the design phase of the software development life cycle where a model gets transformed to a more detailed level. Model refactoring restructures the design to improve the quality and to reduce the complexity of the software designed in terms of model. Refactoring must be raised into model level to improve the structural content while preserving its quality characteristics.

At the level of models, research on refactoring is still in its infancy. Very few tools provide integrated support for model-refactoring as for a lack of proper understanding of model refactoring. Some formalism has been proposed to understand and explore model-refactoring. Van Der Straeten and D’Hondt used a forward chaining logic reasoning engine to support composite refactoring [109]. Biermann, Mens used graph transformation theory as an underlying foundation for specifying model-refactoring [110-111]. Bruno Harbulot attempts to build a link between scientific
programming and software engineering using aspect-oriented programming. Most of these approaches suggest for expressing model-refactoring in a declarative way. This necessitates having a concept on model implementation language which is described in the next section.

7.1.1 Model Implementation Language

Model transformations are used to bridge the gap between high-level, platform-independent and low-level, platform-specific models, close to an implementation. The most frequently used modelling language, which has also gained industrial acceptance, is currently the Unified Modelling Language (UML) [90]. UML is accepted today as a de facto standard for developing software. UML and its sub-language Object Constraint Language (OCL) [91] are regarded as central ingredients of model-centric software production.

The UML comprises a number of different diagrammatic notations for describing certain views of systems. They are the following.

(i) Class Diagram - This is used for a static view of classes and their relationships
(ii) Sequence Diagram - This is used for the scenarios of a system
(iii) State Machines - It is used for protocols of method executions (the dynamic behaviour)
(iv) Component Diagram – This is meant for an architectural view

Other implementation languages usually do not reflect all of these views. They may not have a separation of static view and dynamic behaviour, and may have no means of describing scenarios. Model transformations thus need to move behaviour descriptions from one view to another view, or they may merge two views into one even while still preserving the abstractly specified behaviour.
7.1.2 Model Transformation Rules

Model transformations can be done from one view to another or they can be done from models to code generation. If the models are required to generate code then certain part of the codes are generated manually and certain other part through refactoring. However, this can be achieved only in the following ways as suggested in [31].

(i) Refactor a single model
(ii) Synchronize all related models so that inconsistencies among the models can be avoided
(iii) Few codes need to be regenerated as the model changes
(iv) Manually written codes must be adapted to the newly generated code

Figure 7.1 [31] depicts the above suggested steps.

![Figure 7.1](image)

**Figure 7.1:** Model-driven refactoring

Various formalisms have been proposed to understand and explore model refactoring. Most of these approaches suggest expression of model refactoring in a declarative way. Graph transformation theory [35] was used as an underlying foundation for specifying model refactoring and relied on the formal properties to
reason and analyse these refactoring. Model-refactoring can also be explored by transforming from one view to another view [94]. It can be a transformation of the class diagrams as well. UML class diagrams are widely adopted to help design and visualize software structure. As mentioned in [65], much of the tooling effort is focused on UML models. It lacks tools to express refinements of UML models. Hence, it is necessary to find model based refactoring through UML/OCL.

7.2 Overview of UML/OCL

Class diagrams are manipulated by Object Constraint Language (OCL). OCL is a declarative language for describing the rules that apply to UML. UML, with its various structural and dynamic views, can share many modelling elements. By definition, refactoring should be behaviour-preserving transformations of an application. But one of the problems faced by designers is that it is often hard to measure the actual impact of modifications on the various design views, as well as on the implementation code. The UML also has an advantage in comparison with other design languages: its syntax is precisely defined by a meta-model, where the integration of the different views is given meaning [92]. Therefore, the meta-model can be used to control the impact of a modification, which is essential when it preserve the initial behaviour of an application.

It is intended to show that refactoring can be defined for UML in such a way that their behaviour-preserving properties based on OCL constraints at the meta-model level are guaranteed.

7.2.1 Refactoring using UML/OCL

There are some common refactorings which can be done in model level as listed in [38]. These refactoring are done on UML in which UML are the pre-conditions and there may or may not be impact on the OCL. These are considered to be the rules for refactoring in model level. The refactoring of UML/OCL models is a special type of model transformation and as suggested, can be specified by the Object Management Group (OMG) standard QVT (Query/View/Transformation) [38]. Some of the popular rules, as given in [38] are presented below in table 7.1.
### Table 7.1: Rules for Refactoring in UML/OCL

<table>
<thead>
<tr>
<th>Refactoring rules</th>
<th>Influence on OCL</th>
<th>Pre-condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>RenameClass</td>
<td>No*</td>
<td>UML</td>
</tr>
<tr>
<td>RenameAttribute</td>
<td>No*</td>
<td>UML</td>
</tr>
<tr>
<td>RenameOperation</td>
<td>No*</td>
<td>UML</td>
</tr>
<tr>
<td>RenameAssociationEnd</td>
<td>No*</td>
<td>UML</td>
</tr>
<tr>
<td>PullUpAttribute</td>
<td>No</td>
<td>UML/OCL</td>
</tr>
<tr>
<td>PullUpOperation</td>
<td>No</td>
<td>UML/OCL</td>
</tr>
<tr>
<td>PullUpAssociationEnd</td>
<td>No</td>
<td>UML/OCL</td>
</tr>
<tr>
<td>PushDownAttribute</td>
<td>No</td>
<td>UML/OCL</td>
</tr>
<tr>
<td>PushDownOperation</td>
<td>No</td>
<td>UML/OCL</td>
</tr>
<tr>
<td>PushDownAssociationEnd</td>
<td>No</td>
<td>UML/OCL</td>
</tr>
<tr>
<td>ExtractClass</td>
<td>No</td>
<td>UML</td>
</tr>
<tr>
<td>ExtractSuperclass</td>
<td>No</td>
<td>UML</td>
</tr>
<tr>
<td>MoveAttribute</td>
<td>Yes</td>
<td>UML</td>
</tr>
<tr>
<td>MoveOperation</td>
<td>Yes</td>
<td>UML</td>
</tr>
<tr>
<td>MoveAssociationEnd</td>
<td>Yes</td>
<td>UML</td>
</tr>
</tbody>
</table>

* Rename refactoring influence textual notation of OCL constraints but not their metamodel representation

The first column consists of the names of some common refactoring which are implemented in a modular way. The impact of the refactoring in modular level to OCL is listed in column 2. In many cases, there is no impact in OCL but in few cases, there is impact on the OCL. The rename refactoring influences the textual notation of OCL constraints. In all such modular refactoring, the pre-condition is the UML that signifies that UML diagrams can only be refactored.

The refactoring rule RenameAttribute does not have an influence on attached OCL constraints. More complicated rules that have an influence (e.g. MoveAttribute), are formalized by two QVT rules; one describing the changes in the class diagram and a second that updates the OCL.
Hence, it can be said that the model level refactoring for class diagrams should look for UML and OCL changes.

### 7.2.2 Refactoring AOP

Current research in AOP is focused on problem analysis, software design and implementation techniques. Refactoring in AOP is an area where very little work has been done. Several researchers are trying to explore refactoring in AOP with focus on OO refactoring comparison. In the modular level, class diagrams can be used using UML for AOP but the role of OCL is yet to be explored. Apart from this, in order to improve the quality of refactoring for aspect-oriented software (in code level and in the model level), the following are required.

(i) Tool support for aspect-oriented programs
(ii) Analysis of the effects of refactoring for static and dynamic cross-cutting features (known as introduction and advice respectively)

On the other hand, it has been said that if a class diagram is modified with aspects containing cross-cutting concerns then the behavior may be preserved for some modifications like renaming a variable [19].

Rename variable is a very common Object-oriented refactoring. This refactoring can be used in AOP context. This needs to identify the pre-conditions under which the refactoring may be applied and the actions necessary to apply the refactoring.

Although, renaming variable is a very basic refactoring, it is required to organize it in an aspect-oriented approach. In any program, a variable can be renamed by changing its declaration and by changing all its references. The pre-conditions to this renaming as given in are the following.

1. The new name should not conflict with an already existing variable in the same scope (including in subclasses if the variable is an inherited one).
2. The old name must be replaced with the new one where ever it is currently referenced.
In AOP, the following additional pre-conditions must be introduced if the variable being renamed is a field.

3. The new name does not conflict with a field introduced into the same type (or its subtypes) from an aspect.

Pre-condition 1 ensures that language requirements are honoured. Pre-condition 2 prevents the case where changing a variable reference to use the new name would actually cause it to refer to a new variable, thus changing program semantics. Pre-condition 3 is a straightforward extension of condition 1 to encompass AOP rules. Since, a field introduced into a class from an aspect functions similar to a field declared in the class itself, introduced fields must also be checked for name conflicts.

Assuming the renaming would not violate these pre-conditions, the actions that would be taken to implement the refactoring is now considered. These are the following.

1. The declaration of the variable and all the references to the variable are changed to use the new name.
2. If a field pattern matches this field before renaming and does not match afterwards, it is extended to match.
3. If a field pattern does not match this field before renaming and does match afterwards, it is narrowed to avoid matching.

Action 1 is the normal action for application of this refactoring in any program. Actions 2 and 3 ensure pointcut pattern equivalence. Patterns cannot refer to local variables, so only fields are affected.

It can be said that theoretically renaming variable will work for aspect-oriented refactoring. An AOP refactoring is demonstrated later on.

7.3 Class Diagram Refactoring of Proposed Array Handling

An example to describe the refactoring of a class diagram is now given. The example is the case of a software product-line that is required to be refactored in the design level as
a part of the proposed model. The class diagram refactoring may encounter a situation known as *Composite* design pattern. This example also has a similar situation which applies the same solutions in [36].

The class diagram is modeling a product line that is capable of handling arrays. The product line comprises a class named *array* and has a method *output()*. Array has an attribute *dimension* to represent one or multiple dimensional array. Figure 7.2 depicts the class diagram.

![Class Diagram](image)

**Figure 7.2:** Initial designed class diagram

In Figure 7.2, Product line is a class which has a method *output()*. Array is a class which has a method *operations()* to handle different situations. There is a ‘one to many’ relationship between Product Line and array class.

The class diagram in Figure 7.2 needs some restructuring for the following reasons.

(i) Array is not inherited in Product Line. Hence, separate codes are required for Product Line and Array. These can be avoided by re-organising the class diagrams.

(ii) Composite design patterns are suitable for such situations. Composite design patterns are required where a structure is composed of basic objects that can be recursively grouped in a part-whole hierarchy [36].

This motivates to refactor the class diagram in Figure 7.2. The following steps are used to generate Figure 7.3.

(i) Renaming Product Line class to Group
(ii) Adding an abstract superclass named Product Line to Group
(iii) Making the class Array a subclass of Product Line
(iv) Merging the Group – Group and Group – Array aggregations into Group – Product Line
(v) Finally, move relevant methods and attributes up to Product Line.

Figure 7.3: Refactored Class Diagram

Figure 7.3 is called refactored class (obtained by restructuring of the class diagram in Figure 7.2) because as mentioned in [36], the behavior preservation holds. All the five steps are behavior preserving which is justified.

- Rename refactoring is done. The name ‘Group’ is legal as it is not existing and hence is behaviour preserving.
- The abstract superclass added (Product Line) is empty, as it has neither any attributes nor any methods. This addition will not make any effect. So, the behaviour is preserved.
- Generalization between Array and Product Line is also behaviour preserving as Array had no superclass.
- Merging Group – Group and Group – Array aggregations into Group – Product Line is allowed as these two associations are disjoint, also because they do not have the same objects. Hence, it is behaviour preserving.
• Moving methods and attributes to the superclass Product Line is a case of inheritance.

The advantages of such refactored class diagrams are the following.

a) Separate codes are not required for Product Line and Array.

b) Basic objects of Product line can be recursively grouped in a part-whole hierarchy.

It is also validated that class diagram refactoring can be one of the major approaches for modular refactoring which has been applied in [36] in order to improve the designing of some existing applications.

The above refactoring is platform independent. In order to go for a platform-specific refactoring in MDD one needs to convert a model to a coding level through a programming language.

7.4 Refactored Login Aspect

Login is a very common case of cross-cutting concern. The security system of any software needs this module in a different context. It has already been demonstrated that the proposed security module, which is a cross-cutting concern, can be encapsulated in mixins by AML. This shows that AOP and FOP can be unified to reap the benefits of both and can have a more sophisticated approach for software development. On the other hand, a login module has been developed in AOP in Chapter 3. The refactoring in a module of AOP which is a security aspect is now evaluated. This is required for testing a module through RFDs. The module can be well visualized by depicting the processes and by tracing the inconsistencies. This becomes mandatory if the module needs to be integrated to any existing software. This type of unit testing in model level needs RFDs. It may be called as ‘dry run of modules’.

7.4.1 Aspect Refactoring

Login is very often mentioned as an example of a cross-cutting concern. However, this is the case only if the login functionality occurs as a secondary concern. This happens,
when the login functionality is referred consistently by a number of elements (e.g., methods) which can be captured through an aspect definition (e.g., a pointcut).

The example [68], given below shows a consistent use of login over the whole system which prints the names of the methods throwing exceptions. A typical implementation of the same concern in OOP would consist of calling the login method from all the methods throwing exceptions. This situation can be described as scattered method calls. This situation can be handled well if instead of login the exception, they are wrapped into another exception to be thrown. The refactoring would be similar to the aspect solution for login that consists of application of a pointcut and an advice mechanism.

```java
public aspect ExceptionLoggerAspect {
    
    pointcut exceptionLogMethods() : call(* *.*(..)) && !within(ExceptionLoggerAspect);
    after() throwing(Throwable ex) : exceptionLogMethods() {
        
        Signature sig = thisJoinPointStaticPart.getSignature();
        log("Exception logger aspect [" + sig.getDeclaringType().getName() + "." + sig.getName() + "]");
    }
}
```

The aspect calls the exceptionLogMethods() with wild cards and uses the after advice i.e. the exception will call the method only after any exception is triggered. The joinpoint is used to invoke the method. This exception is thrown, after the occurrence of a certain event which can take place through AOP and not through OOP.

This example has been chosen to show that AOP can also be used for exception logger aspect, which is required to be added in the security module. This example also shows that there is a need to refactor from one approach to another approach. The author has written the codes in AspectJ which have been used to demonstrate the significance of RFD in the next section.
7.5 RFD in Exception Log Module

The exception logger aspect has been demonstrated in code level. It has also been shown [69] that it is a better approach to throw exception in a situation like logging. It is now shown that if this is improvised in the security login then this is a case of modular refactoring where appropriate action will be taken as and when required without changing the functional-behaviour of the proposed model.

7.5.1 Different Levels of RFD

The following diagram shows the level 0 RFD of the refactoring process to introduce the Exceptionlogger aspect which has been identified to be refactored.

![RFD Diagram](image)

**Figure 7.4:** Level 0 RFD of Exceptionlogger

It has been proved by the authors of [69] that the exception designed by them (as shown in previous section) is a cross-cutting concern and so they are encapsulated in an aspect. Hence, exception handling concept has been used directly to the present RFD to show that it leads to pure refactoring.

The processes for designing the level 0 RFD are the following.

(i) Identification and design of exception module codes for refactoring
(ii) Application of graph transformation on the identified codes which have to be refactored in order to check for pure refactoring (i.e., it leads to isomorphic graph)

(iii) Execution of the refactored codes

(iv) Result are interpreted as behavior preserving

The next level RFD is the Level 1 which is identifies the exception throw module and describes the sequence of operation. Figure 7.5 depicts the details.

**Figure: 7.5: Level 1 RFD of Exceptionlogger**

The Level 1 RFD consists of two processes which provide the following informations.

(i) Exception throw module has been identified

(ii) The methods of the exception throw are re-designed

The next level of RFD is Level 2 which describes the details of Graph transformation. The next diagram is demonstrates the Level 3 RFD which describes Exception Logger Aspect and comprises of pointcut, joinpoint and suitable method call. This is followed by termination of login for an unauthorized person or continuation for the login process for an authentic person.
The Level 3 RFD is the refactored code that comprises the following.

(i) A pointcut for exceptionLogMethods() is called with wild cards and is executed within the aspect ExceptionLoggerAspect.

(ii) The exceptionLogMethods() is invoked through a joinpoint after an exception occurs. This is an attempt to login.

(iii) The method takes user’s identification details.

(iv) Authenticity is tested, after which a user gets a login or is refused for a login with a termination.

The last level describes each and every detail of the login process to verify user’s authenticity. This aspect module is better than an object module because it throws exception after a certain event occurs.

The RFD not only ensures for pure refactoring detection but also depicts all the details of each process. This designing of flow diagrams helps in the later stage for designing the modules for refactoring. RFD can be used in code level or in model level to enhance the power of detecting impure refactoring.
7.6 Case Study: Smart Telephone Network Product Line

The modular refactoring is also required for the present case of STNPL at the designing phase. The designing phase may use different diagrams to represent it in models. It is aspired to design user to server interaction through class diagrams. As mentioned in section 7.3, composite design patterns can be encountered in class diagrams, which is one of the major class diagram refactorings. This case study needs similar refactorings in order to cope up the composite design patterns. In the next section, few class diagrams are designed. They are required to depict the interaction among user, PSTN, gateway and Server for a telephonic conversation.

7.6.1 Class Diagrams of STNPL

The class diagram uses the classes PSTN and Server. Gateway is an attribute of the class server. The operations are Create Connection, User Interaction and Call Termination. The initial class diagram is given in Figure 7.7.

![Initial Class Diagram of STNPL](image)

**Figure 7.7:** Initial Class Diagram of STNPL

The class diagram is modelling the STNPL. It is required to refactor the class diagram in order to remove the following flaws.
(i) Instead of two separate classes PSTN and Server, Server is made as a sub class of PSTN. This will give a better relationship between PSTN and Server

(ii) Inheriting the methods of server to PSTN

(iii) Composite design patterns are seen here

The refactoring of Figure 7.7 class diagram is carried out by executing the following steps.

(i) Renaming PSTN to Telephone

(ii) Adding an abstract super class named PSTN to Telephone

(iii) Making the class Server a sub class of PSTN

(iv) Merging Telephone – Telephone and Telephone – Server aggregations into Telephone – PSTN

(v) Finally, moving the relevant methods and attributes up to PSTN (Inheriting the methods and attributes)

![Refactored Class Diagram of STNPL](image)

**Figure 7.8:** Refactored Class Diagram of STNPL

As explained in section 7.3 such refactorings are pure since they are behavior preserving, so the refactored STNPL in Figure 7.3 is a pure refactoring with better designing.
7.6.2 Relevance of Class Diagram Refactoring in STNPL

Model driven development needs class diagrams, activity diagrams, flow diagrams, state diagrams etc. in designing phase. The designed modules need to be modified either to remove the flaws in the model or to upgrade the model. The modifications should not change the external behaviour of the models. This implies that pure refactoring is required. In order to judge a refactoring to be pure in model level, some standard refactorings are done. As such class diagram refactoring has been done in [36]. So, the similar refactorings are applied in our case study also. This leads to the following observations, which can be called as the steps required for class diagram refactoring.

(i) Initial class diagram needs to identify the classes
(ii) The relationships among the classes are analyzed
(iii) If composite design patterns are not present then simple refactoring is done by associating the classes
(iv) If composite design patterns are present, the steps mentioned in [36] are followed which gives pure refactoring

The advantages of class diagram refactorings are the following.

(i) Easy to identify and design the classes
(ii) Relationship among the classes can be analyzed (deciding super class and sub class)
(iii) If required new classes (super or sub classes) can be introduced
(iv) For any changes in the classes (as done in (ii) and (iii)) testing is required for behaviour preservation

It is found that any designing in class diagram can be modified, provided it is not impure. Hence, class diagrams must be incorporated in modular designing approach as restructuring of the classes can be done by following the steps mentioned above. This modular refactoring can be used in MDD to minimize impure refactoring and also to obtain better designed classes through refactoring the existing classes.
7.7 Summary

Modular refactoring is obtained by refactoring the class diagrams and aspect modules. The class diagram refactoring is done for a designed class diagram, capable of doing some array operations and in a class diagram of the case study. Composite design patterns are found in both the class diagrams. Refactoring has been done the class diagrams. These refactorings help to find the solutions for modular refactorings in class diagrams. On the other hand RFDs are applied in an existing aspect module to depict the processes, flow and in tracing the inconsistencies in the module. The RFDs in different levels have suggested being pure refactoring. Based on this result, one can integrate this module in the security aspect. This justifies that individual modules can also be tested to avoid impure refactorings, before integrating any module to an existing software. The refactoring techniques used either in class diagrams or in any existing module need to be incorporated to make MDD a powerful mechanism of software development.