Chapter 6

Techniques for Detection of Impure Refactoring

Several techniques and tools have been developed to assist refactoring either in the designing phase or in the maintenance phase. However, the current tool support is not adequate as the refactoring tools are facing several challenges such as composition of refactoring, searching refactoring, planning refactoring sequences, computation of conflicts and dependencies of refactoring etc. These challenges would greatly benefit from a formal support provided by techniques which can detect impure refactoring. In this chapter, some approaches have been suggested for early detection of impure refactoring. Integrating this approach in a refactoring tool is proposed and this can be considered as the first step of refactoring models. Refactoring Flow Diagram (RFD) has been named. Modular refactoring can be done more successfully by applying RFD in model levels. The proposed RFD has been tested in a refactoring tool called KABA and the functioning of RFD in different levels has been explained. RFDs are also used in the case study to demonstrate their utility in any product line.

6.1 Background and Motivation

Software development approaches experienced a shift from the classical waterfall model, (where analysis is fully completed before design and design is fully completed before implementation) to evolutionary approaches. In this approach, a simple version of what is required is extended iteratively to build a more complex system. So, program restructuring is an important stage during each evolutionary phase. As a consequence, refactoring has been identified as central to software development and software maintenance, especially within Software Engineering (SE) communities. The main objective of refactoring is to improve the design by either reducing coupling and/or
increasing the cohesion. The improvement is done either on some existing codes or on some existing models.

Initially, refactoring had been typically done manually or with the help of text editors with search and replace facilities. Manual refactoring is tedious, error-prone and costly. It depends on extensive testing to ensure that functionalities are preserved. While testing can show up bugs and improve confidence, it cannot prove correctness. Therefore, refactoring tools for various programming languages, which could help programmers perform refactoring automatically, and could be proven functionality-preserving, were highly desirable. Since that time, several refactoring tools came into existence.

Refactoring cannot be done efficiently without any (semi) automated tool, which not only assists in refactoring but also assists in detection of external functional changes. The detection of external behavioral changes implies a faulty refactoring. In this situation the tool must undo and do a rollback to the previous state in order to deactivate the refactoring and preserve the original behavior. The major reason for such failure is identification of wrong codes for refactoring which is one of the prime challenges the refactoring tools are facing. This is the main motivation to improve the refactoring process.

In order to improve the refactoring process, standard tools are required to detect the impure refactoring in an early state. Several approaches have been developed to detect impure refactoring including the graph search methods proposed by Shinpei and by Perez and Crespo [26, 27]. On the other hand, Emerson [28] proposed a model which depicts the conventional refactoring process. This model shows the sequence of the refactoring processes but it has no role in early detection of impure refactoring. If Emerson’s model is been revised with Graph search methods then it can be used as a tool for early detection of impure refactoring. This has motivated to develop a similar tool, as described in Section 6.4.
6.2 Classifications of Refactoring and Refactoring Techniques

Refactoring is considered to be of different types. There are two dimensions in which the refactoring can be classified. These dimensions are based either on the level of refactoring or on perfection basis. Based on level of a refactoring, it is classified as follows.

   a) Simple
   b) Compound Refactoring

In the context of perfection, it is classified as Pure and Impure Refactoring.

Though the principles of refactoring suggest for simple refactoring but sometimes refactoring is performed not as an individual operation but also as a part of compound operation. For example, one refactoring is dependent in such a way on another refactoring that it becomes effective only when the other is performed. As compound refactoring is more frequently used in software modifications, its behavior preservation may not be achieved easily. Such situations are known as impure refactoring. It can be said that compound refactoring is more prone to impure refactoring.

Though refactoring is done either in the code level or in the model level, but this work mainly deals on the refactoring techniques that are capable of detecting impure refactoring. Different refactoring techniques are there in order to detect the impure refactoring. Out of these techniques, the graph search [26] and graph transform methods [27] are considered to be successful ones.

Shinpei’s graph search technique [26] considers the following things.

   (i) A version of a program as a state.
   (ii) Refactoring as a transition operator.
   (iii) Searches for a path that approaches from the initial state to the final state.
   (iv) The efficiency of the search is improved by focusing on the differences in the source code between both the current and the final state. It facilitates the
candidates of refactoring to be applied next and estimating the heuristic
distance to the final state.

(v) The authors have modelled the detection as best-first graph search problem.

Another similar technique was proposed by Perez and Crespo [27]. The refactoring is
handled by Graph transformation and the approach is the following.

(i) Programs and refactoring are represented with graphs in a language
independent way. Programs are represented with graphs and refactoring
operations are described as graph transformation rules.

(ii) The abstract syntax trees are used.

(iii) Graph parsing algorithms are used to find the refactoring sequence between
two versions of a software system.

With Graph transformation approach [27], one has to identify the following.

a) The original/old system as a start state.

b) Refactoring operations as state changing operations (edges).

c) The refactored/new system as the goal state.

d) Whether a refactoring sequence exists by tracing a path from the start state to the
goal state. The refactoring sequences can be obtained through a basic search
algorithm.

### 6.2.1 Refactoring Sequences

A refactoring definition is given by using pre-conditions and post-conditions. The start
state is repeatedly modified through application of graph transformation rules. Those
refactorings are selected for iterations whose pre-conditions are held in the current state
and post-conditions are held in the goal state. When refactorings are not selectable any
more, the algorithm backtracks to the last transformation applied. The algorithm
terminates with success when the current state graph is isomorphic to the goal state
graph. The path found to the goal state is the refactoring sequence. When the current
state graph and goal state graph are not isomorphic, the algorithm terminates with a
failure.
The main advantage of this refactoring discovering method is that it can be used for detecting the possibilities of refactoring. This detection has to be done as the first step. This method supports few well known refactoring like pullUpMethod, renameMethod, removeMethod and removeClass [64]. It is highly required to make a comparative analysis of the two techniques.

6.2.2 Analysis of the Refactoring Techniques

The two techniques [26, 27] as given by the authors have some similarities and dissimilarities. The similarities are that both of the techniques rely on searching algorithms and detecting impure refactoring. The dissimilarities are the following.

a) The first technique can handle both simple and compound refactoring whereas the second is capable of handling only simple refactoring.

b) *Best-first graph search* (as termed by the authors [26]) is used in the first case whereas *depth-first* is used by the second.

c) The first technique considers program as states and the refactoring as transition operators whereas the second technique considers program as a graph and refactoring as a graph transformation.

From the above analysis it can be said that the first technique is better in context of supporting both simple and compound refactoring. However, the second technique is better in all other contexts including a well known graph search method (i.e., depth-first search method), presentation of program as graph and refactoring as graph transformation etc. Hence, incorporation of the second technique in the present proposed model has been considered.

6.3 Emerson’s Model for Conventional Refactoring Process

In order to cope with the challenges of early detection of impure refactoring, Emerson [28] proposed a model which demonstrates the traditional refactoring. The author depicted the conventional refactoring process in the model, shown in Figure 6.1.
The above model has the following steps as mentioned by the author [28].

**Identify:** Finding code to be refactored which is often done by the programmers manually.

**Select:** The selected codes are placed into an editor.

**Initiate:** Desired refactoring is chosen either from a menu or hotkey.

**Configure:** Some options are chosen from a dialog box for the desired refactoring.

**Execute:** Perform the refactoring by clicking the OK button.

**Interpret Results:** Determine whether the refactoring obtained is the desired one or not.

**Interpret Errors:** Take suitable actions when pre-conditions are violated.

**Undo:** Revert back to the program when some unexpected results obtained.

**Refactor recursive:** To make the desired refactoring successful, recursively perform sub-refactoring.

This model has the following limitations.
(a) The model shows a traditional refactoring only. It has no role in improving refactoring processes except for providing an undo process on account of improper refactoring.

(b) Recursive refactoring works only when the refactoring tool does not work properly. This happens when wrong configurations are obtained or the initiation is not working etc.

(c) The model only shows the steps required for refactoring using a refactoring tool. It has no provision to detect impure refactoring at an early stage for e.g. immediately after code selection.

Hence, it is needed to improve the model for providing a better refactoring technique which eliminates the problems mentioned above. Apart from that, if in this model the graph transformation rule has been incorporated to find out whether a refactoring sequence exists or not, then recursive refactoring can be avoided.

As suggested in [27], if there is a path between the original state and the goal state then a refactoring sequence exists. Hence Emerson’s model [28] is revised with some of the features mentioned above and it is termed as the first level of refactoring. This is followed by several refactoring levels provided the first and the successive levels indicate a pure refactoring. So, in the present work, refactoring model consists of several levels which give a detail description of the refactoring activities and also traces any impure refactoring in any level. It is started with a 0 level to depict the overall refactoring process prior to level 1.

### 6.4 Proposed Model for Early Detection of Impure Refactoring

A revised model of Emerson is now proposed which incorporates the graph transformation rules as the initial step to detect impure refactoring. The model consists of several steps showing the diagrammatic flow of pure refactoring. We name it as **Refactoring Flow Diagram (RFD)**.

The main purpose of RFD is as follows.

(i) Find whether a continuous flow exists or not.
(ii) Depict the overall process and the detailed processes.

(iii) Trace an impure refactoring through existing techniques.

RFD consists of several levels in which zero (0) level is the starting level which shows the overall refactoring activities. Level 0 RFD begins from Start state. The different processes are:

(i) **Identify**: This step is responsible for choosing the codes/models to be refactored.

(ii) **Apply Graph Transformation**: As mentioned in [27] the graph transformations are applied.

(iii) **Exit/Execute**: Exits when the current state graph is not isomorphic with goal state graph otherwise refactoring is done.

**Figure 6.2**: Refactoring Flow Diagram (Level 0)
(iv) **Interpret Results:** This process checks for the refactoring status. If the refactoring is behavior preserving (pure refactoring), it proceeds to the next process to check whether more refactoring required.

(v) **Undo:** If the refactoring leads to an impure one, the process is executed which is followed by the Identification process.

(vi) **More Refactoring:** This process checks, whether the refactoring is to be continued or not. The former leads to identification process whereas the later leads to Clean up process.

(vii) **Clean up:** This process leads to the termination of refactoring session by winding up the refactoring environment.

The main objective of the model is to depict each and every process of refactoring with its sequence of occurrence. This reveals the clarity of the entire refactoring. In order to enhance the clarity, the entire refactoring process is broken up into different levels. The starting level, which depicts the overall refactoring, is termed as level 0. The successive level gives the detail of the processes. The level which consists ‘Apply graph transformation’ is not shown in any of the cases as it is known that it will either give an isomorphic or non-isomorphic between the current state and goal state, which indicates a pure or impure refactoring respectively.

The level 0 RFD is the first level of RFD which gives an overall idea of the Refactoring process. The detail refactoring must be presented at the subsequent levels. For example, Level 1 RFD, which comes after the above mentioned Level 0 RFD, may be the identification of codes or may be identification of models which requires refactoring. During this level, the pre-conditions must be defined and analyzed. Similarly, the next level can be the application of the Graph transformation rules. This level will detail the graph transformation steps. This way of depicting the refactoring processes in details will minimize the faulty refactoring and will also help to give a holistic view of the entire process.
6.5 Overview of KABA

KABA is an automated refactoring tool developed by Snelting and Streckenbach [29]. KABA refactoring tool can refactor class hierarchy. The basic characteristics of KABA tool is listed below.

- **a)** KABA analyses a class hierarchy and refactors the hierarchy in such a way that data members and methods operating on the data are always grouped together at the cost of preserving behaviour.
- **b)** KABA refactoring improves cohesion. It is implemented in form of a refactoring browser [29].
- **c)** KABA is a research prototype. It can handle full Java (without reflection). It has been applied to several small and medium-scale systems, and has generated useful refactoring. More details about the KABA refactoring browser, semantics-preserving class merging and elimination of multiple inheritances can be found in literature [87-89].

6.5.1 Applying Proposed RFD on KABA

The sample program (Figure 6.3) written in pseudo-language has been considered in which the original class is divided when different clients use different subsets of the functionalities of a class. This example of class hierarchy has been refactored in using KABA tool and a successful refactoring has been obtained [29]. Since this gives a successful refactoring, the RFD in this example has been applied.

```java
class A {
    int x, y, z;
    void f() {y = x;}
}

class B extends A {
    void f() {y++;}
    void g() {x++; f();}
    void h() {f(); x--;}
}

class Client {
    public static void main(String[] args) {
        A a1 = new A(); // a1
        A a2 = new A(); // a2
        B b1 = new B(); // b1
        B b2 = new B(); // b2
        a1.x = 17; a2.x = 42;
        if (...) { a2 = b2; }
        a2.f(); b1.g(); b2.h();
    }
}
```

**Figure 6.3:** Sample Program
Here, Class A consists of member data x, y and z and member function f(). Class B extends Class A and inherits members of class A with method extensions of f() and definition of methods g() and h(). The client objects are a1, a2, b1 and b2.

The following observations have been made.

(i) Object a1 accesses only ‘x’. But object a2 accesses f() and hence ‘y’ also. So, class A can be divided into two sub-classes.

(ii) Object b1 calls g() and object b2 calls h() and so class B can be split.

All these divisions of classes do not change the behavior of the program output and so this refactoring is a pure refactoring. Hence, the RFD of the above KABA refactoring is now applied.

![Diagram](image.png)

**Figure 6.4:** Level 0 RFD

Figure 6.4 depicts the following.

(i) It identifies the class hierarchy

(ii) It applies the graph transformation to show that current state graph is isomorphic with goal state graph

(iii) Refactoring executed

(iv) The result interpreted as the refactoring is behavior preserving and the cohesion has improved, due to KABA techniques

(v) It has terminated with cleaning up
The Level 0 RFD as mentioned earlier, gives the overall process of refactoring. Since the example is tested as successful refactoring in [29] so the options like failure process or undo process (as given in our original model) has not been included. The example does not need to refactor further codes and hence more refactoring process has also not been shown here.

Level 1 RFD is now explored which represents the pseudo-code of the sample program (Figure 6.3) to demonstrate the identification of the class hierarchy (in level 0 RFD) and to divide the classes with their respective members.

![Level 1 RFD Diagram](image)

**Figure 6.5: Level 1 RFD**

Level 1 RFD (Figure 6.5) is suggesting for identifying the hierarchical class relationship with member distribution. Level 2 RFD is applying the Graph transformation on the classes identified, that come up with an isomorphic graph. This has not been presented here as it is known that the graph transformation process detects impure refactoring.

Hence, the final refactoring can be executed. The outcome of the refactoring is in Figure 6.6, depicting the Level 3 RFD [29] (which is the final level in this example).
Level 3 suggests that Class B can have two sub-classes B1 and B2. The member functions g() and h() are inherited in B1 and B2 respectively. The KABA tool has obtained this refactoring and it is behavior preserving. Before using the KABA tool, the RFDs may be used in different levels. This indicates that RFDs must be used as a first step for refactoring before applying any tools.

In lieu of the above discussion, it can be said that refactoring processes must be analyzed first before its application. As a part of the analysis, application of RFD is suggested in order to detect impure refactoring. This may reduce the probability of impure refactoring. In the next section refactoring with RFD is applied in the present Case study.

6.6 Case Study: Smart Telephone Network Product Line

The STNPL consists of few features and aspects as explained earlier. The sequence diagram of PSTN (Figure 4.7) shows few operations, of which

(i) Put down receiver    (ii) Connection closed    and (iii) Call ends need some modifications. This modification must not change the behavior of the system. Hence, the modification must be behavior preserving. Other way it is called as behavior preserving refactoring. These three functions are part of connection termination process.
6.6.1 Refactoring of Connection Termination Process

Put down receiver implies connection to be closed with call ending. So, the method connection closed is moved to the method call ends. A MoveMethod refactoring is applied. In order to do this, the operations, pre-conditions and mechanisms are identified which are as follows.

**Operations:** Source class identification, Target class identification and identification of codes in the methods to be moved

**Pre-conditions:** The target class must be able to accommodate the codes and name conflicts must not arise

**Mechanisms:** RFDs have been applied as an early detection of impure refactoring

It is now assumed that the identifications of source class, destination class and codes have been done. The graph transformation finds the fulfillment or non-fulfillment of the pre-conditions leading to pure or impure refactorings respectively. So, RFDs are required to depict the whole process of refactoring.

The level 0 RFD is given in Figure 6.7.

![Figure 6.7: Level 0 RFD of STNPL](image)
Figure 6.7 shows the following processes.

a) Move *connection closed()* to *call end* class.

b) Graph transformation is performed and current state graph is compared with goal state graph.

c) If the two graphs obtained through graph transformation are not isomorphic then the MoveMethod refactoring is impure. This situation arises when the pre-conditions are not satisfied. The refactoring should be aborted and identification operations must be revised.

d) When the two graphs obtained through graph transformation are isomorphic then the MoveMethod refactoring is pure which allows for execution.

e) The result is interpreted to be behaviour preserving, confirming the pure refactoring with a termination of RFD.

The next level, which is Level 1, gives the detail of source, destination and code identification operations in Figure 6.8.

![Diagram](image)

**Figure 6.8:** Level 1 RFD of STNPL
Level 1 RFD depicts the following.
   a) Source class is Connection
   b) Destination class is Disconnection
   c) All codes and data of the method `connection closed()` is highlighted
   d) Graph transformation is applied

The last step which suggests for applying graph transformation gives Level 2 RFD. As Level 2 is beyond the scope of this thesis, so the higher level RFDs are continued with two possibilities, original state graph is isomorphic with goal state graph (i.e. pure refactoring) and original state graph is not isomorphic with goal state graph (i.e. impure refactoring). In the former case, Level 3 is required whereas in the later case, the refactoring must be aborted.

The last level in this case is Level 3 which is shown in Figure 6.9.

![Figure 6.9: Level 3 RFD of STNPL](image)
The process execution suggests the following.

a) The codes and data of the method `Connection closed()` in source class are removed to the destination class using suitable tools of an editor

b) Results are verified

c) Behaviour preservation is detected

d) Refactoring granted through termination

The RFDs in different levels not only detects impure refactoring before applying it but also gives a detail picture of the steps to be carried out in the refactoring process. This detail designing can be applied as the first stage of refactoring which can be considered as an analysis phase of refactoring.

6.7 Summary

A concept of Refactoring Flow Diagram is introduced in this chapter. It not only helps in early detection of impure refactoring but also shows the flow of refactoring processes. This enables one to synchronize and sequence the processes to give a holistic view of refactoring process. The higher level RFDs may incorporate both simple and compound refactoring. A refactoring must start with RFD which is followed by the implementation of refactoring through refactoring tools. Emerson’s model, demonstrates how people use refactoring tools. The model has further been modified to present the flow of refactoring in terms of RFD. A study has been made on the existing refactoring tools like KABA and has been applied in a sample program, which had been refactored successfully by KABA tool. The proposed RFDs worked on the program yielding different levels of RFD. The RFDs can be used as the first step of refactoring to avoid or minimize impure refactoring. The refactoring performed in the present case study further suggests that RFDs can be used as tool for analyzing the possibilities of refactoring. Since software upgradation needs refactoring either in code or in model level so RFDs can be very helpful tool in SDLC. MDD techniques can become more efficient by using RFDs in context of refactoring.